



Calhoun: The NPS Institutional Archive
DSpace Repository

Theses and Dissertations

1. Thesis and Dissertation Collection, all items

1977-03

**MICRO-COBOL: an implementation of Navy
standard HYPO-COBOL for a
microprocessor-based computer system**

Craig, Alan Scott

Monterey, California. Naval Postgraduate School

<http://hdl.handle.net/10945/18198>

This publication is a work of the U.S. Government as defined in Title 17, United States Code, Section 101. Copyright protection is not available for this work in the United States.

Downloaded from NPS Archive: Calhoun



Calhoun is the Naval Postgraduate School's public access digital repository for research materials and institutional publications created by the NPS community. Calhoun is named for Professor of Mathematics Guy K. Calhoun, NPS's first appointed -- and published -- scholarly author.

**Dudley Knox Library / Naval Postgraduate School
411 Dyer Road / 1 University Circle
Monterey, California USA 93943**

<http://www.nps.edu/library>

P86 - CLOSE' Format shall be close file-name -
P96 - OPEN " " " OPEN { } .

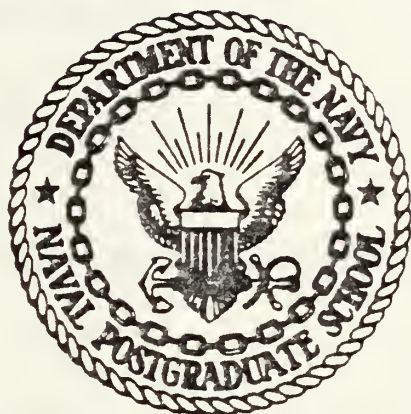
DI = Duplicate IO

MO = Memory Overflow.

SCR Routine screwed up decode calls for 9, Routine
generates 4 bytes.

NAVAL POSTGRADUATE SCHOOL

Monterey, California



THESIS

MICRO-COBOL
AN IMPLEMENTATION OF
NAVY STANDARD HYPO-COBOL
FOR A MICROPROCESSOR-BASED COMPUTER SYSTEM

by

Alan Scott Craig

March 1977

Thesis Advisor:

Gary A. Kildall

Approved for public release; distribution unlimited.

T178074

REPORT DOCUMENTATION PAGE

1. REPORT NUMBER		2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) MICRO-COBOL an implementation of Navy Standard Hypo-Cobol for a microprocessor-based computer system		5. TYPE OF REPORT & PERIOD COVERED Masters Thesis; March 1977	
7. AUTHOR(s) Alan Scott Craig		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		8. CONTRACT OR GRANT NUMBER(s)	
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		12. REPORT DATE March 1977	
		13. NUMBER OF PAGES 170	
		15. SECURITY CLASS. (of this report) Unclassified	
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.			
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)			
18. SUPPLEMENTARY NOTES			
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) COBOL, compiler, formal grammar, microprocessor, microcomputer, LALR(1), HYPO-COBOL			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) A compiler for ADPESO standard HYPO-COBOL has been imple- mented on a microcomputer. The implementation provides nucleus level constructs and file options from the ANSI COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. The language was implemented through a self-hosted compiler and run-time package			

on an 8080 microcomputer-based system. Both compiler and interpreter can be executed in 12K bytes of user storage.

Approved for public release; distribution unlimited

MICRO-COBOL
an implementation of
Navy Standard Hypo-Cobol
for a microprocessor-based computer system

by

Alan Scott Craig
Captain, United States Marine Corps
B.S., Brigham Young University, May 1971

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
March 1977

ABSTRACT

A compiler for ADPES0 standard HYPO-COBOL has been implemented on a microcomputer. The implementation provides nucleus level constructs and file options from the ANSI COBOL package along with the PERFORM UNTIL construct from a higher level to give increased structural control. The language was implemented through a self-hosted compiler and run-time package on an 8080 microcomputer-based system. Both compiler and interpreter can be executed in 12K bytes of user storage.

CONTENTS

I.	INTRODUCTION.....	7
A.	HISTORY OF COBOL.....	7
B.	MOTIVATIONS OF HYPO-COBOL.....	8
C.	MICROCOMPUTERS.....	9
1.	Hardware.....	9
2.	Software.....	10
D.	OBJECTIVES OF MICRO-COBOL.....	11
II.	MICRO-COBOL MACHINE.....	13
A.	GENERAL DESCRIPTION.....	13
B.	MEMORY ORGANIZATION.....	16
C.	MACHINE OPERATIONS.....	16
1.	Format.....	16
2.	Arithmetic operations.....	16
3.	Branching.....	17
4.	Moves.....	21
5.	Input-output.....	24
6.	Special instructions.....	28
III.	MICRO-COBOL IMPLEMENTATION.....	30
A.	COMPILER IMPLEMENTATION.....	30
1.	General method.....	30
2.	Control flow.....	31
3.	Internal structures.....	32
4.	Part one.....	33
5.	Part two.....	41

B.	INTERPRETER IMPLEMENTATION.....	50
1.	General structure.....	50
2.	Code modules.....	50
a.	Arithmetic instructions.....	51
b.	Branching.....	52
c.	Input-output operations.....	52
d.	Moves.....	53
3.	Limitations.....	54
C.	SOFTWARE TOOLS.....	54
IV.	CONCLUSIONS.....	56
	APPENDIX A - MICRO-COBOL USERS MANUAL.....	58
	PROGRAM LISTINGS.....	115
	LIST OF REFERENCES.....	168
	INITIAL DISTRIBUTION LIST.....	170

I. INTRODUCTION

A. HISTORY OF COBOL

As indicated in the name, COBOL - COmmon business ORiented Language - was intended to be a common standard computer programming language with consistent implemen- tations on various machines. Backed heavily by the Department of Defense, COBOL has become a widely accepted language for data processing applications. Over the fifteen years of its existence the language has undergone several revisions and still continues to be upgraded and changed [1].

The evolution of COBOL has resulted in a large language containing numerous capabilities, many of which are not appropriate for a given machine nor desired by a class of users. For this reason the COBOL language is broken down into modules which may be implemented at various levels. The minimal standard COBOL, as currently defined, contains only the lowest levels of three modules out of the possible twelve modules which currently exist.

B. MOTIVATIONS OF HYPO-COBOL

None of the existing standard sets of COBOL modules fit the requirements of the Department of the Navy, and thus HYPO-COBOL was developed. Rather than taking one of the implementation levels described in the standard, another subset of the complete instruction set was developed which includes only parts of modules. HYPO-COBOL was designed to impose minimal requirements on a system for compiler support. Where possible, short constructs were used in the place of longer ones. Where multiple reserved words serve the same function in COBOL, the shortest form was used. There is no optional verbage in the language, and there are no duplicate constructs performing the same function.

Limits were placed on all statements that have a variable input format so that all statements have a fixed maximum length. Where possible, such constructs were removed completely from the language. In addition, user defined names were limited to twelve characters to reduce symbol table storage requirements.

Rather than include the standard levels of implementation for all of the modules, constructs were included only as required. In addition to low level constructs, the PERFORM UNTIL construct was included to allow better program structure. Further justification for the manner of subset-ting and a highly detailed description of each element of the language is contained in the HYPO-COBOL Manual [10].

C. MICROCOMPUTERS

Current technological advances in the design of integrated computer components have lead to the proliferation of single chip central processors known as microcomputers. The number of chips produced and the varying capabilities of each product make generalizations very difficult. The term microcomputer, however, is generally used to describe a system built around one of these processors. Such a system would have memory, input and output capabilities, and timing circuits as well as a central processor. One chip systems with all of these capabilities are currently becoming available.

1. Hardware

The most significant factor in the proliferation of microcomputer-based systems has been their cost. Reasonably powerful central processors can currently be purchased for less than twenty dollars, resulting in the appearance of many new applications. Along with the low cost of the central processor have come low cost peripheral devices that are well suited to the speeds and capabilities of the microcomputers. In the case of traditional users of computers, the low cost of microcomputer hardware has led to new uses and to distributed processor networks. Changes in the cost and capabilities of microcomputers have been dramatic over the last several years, with more and more capabilities being offered at lower prices.

2. Software

Software has lagged far behind the developments in hardware for microcomputers. Most of the currently available systems do not support high level languages at all, and where supported, the languages are often systems languages rather than applications oriented languages. One of the restrictions imposed by many high level languages has been the requirement for cross-compiling on a more powerful machine [7]. In addition, some of the resident compilers require large amounts of memory. Recent work on versions of BASIC however, has led to quality resident compilers for scientific type calculations [6].

To allow the use of microprocessor systems in many of the proposed applications, languages need to be developed that will run on microcomputers without placing unreasonable demands on their capabilities and size. If the developments in hardware continue at their present rate, software will almost certainly continue to lag behind. However, current compiler construction techniques do seem to make it possible to provide the required languages, at least on the current types of hardware [3].

D. OBJECTIVES OF MICRO-COBOL

The major objective of this project was to implement HYPD-COBOL on an 8080 microcomputer-based system. As steps toward that objective, the following underlying goals were established: first, define HYPD-COBOL as an LALR(1) grammar [12]. Second, construct a compiler based on a table-driven parser for that LALR(1) grammar. Third, implement an interpreter to run the intermediate language instructions produced by the compiler.

While it was recognized that there would be difficulties in displaying the complete capabilities of the HYPD-COBOL language on the equipment currently available at the Naval Postgraduate School, it was considered feasible to implement a major portion of the subset with the current equipment and software.

One of the justifications for this project was the current standard policy of the Department of Defense to require all computers used in non-tactical environments to be capable of executing COBOL. In the case of the Department of the Navy, the standard that would need to be met for a microcomputer-based system is HYPD-COBOL.

Finally, it should be noted that there was no attempt to add to the HYPD-COBOL definition. One area of investigation was to test the feasibility of the subset. In defining the grammar, areas were found where additions could have been made, and future users may require enhanced capabilities to

make the language fit their requirements. Indications have been made, in the following sections, of places where changes seemed appropriate.

.

II. MICRO-COBOL MACHINE

A. GENERAL DESCRIPTION

The following sections describe the MICRO-COBOL pseudo-machine architecture in terms of allocated memory areas and pseudo-machine operations. The pseudo machine was the target machine for the compiler and was implemented through a programmed interpretation. The MICRO-COBOL machine has been given first, since all other system components can be described in terms of the target machine.

There were several ways to design the pseudo machine. The parser used produces operations in the order convenient for a stack machine, and other applications have used a simulation of a stack machine to interpret the output of the compiler [6]. The operations required for HYPOL-COBOL did not require the use of a stack but could be designed as relatively independent operations. It would be possible to produce an interpreter that consisted of a set of subroutines which would be called directly by machine level operations on the 8080. The emitted code would then consist of instructions to load parameters and calls to the subroutines. This second idea was rejected due to the limited time available for the production of the project and because the code generation would then be very closely tied to the exact implementation of the interpreter. It was de-

cided to produce output code for a pseudo machine that would be defined to have all of the needed operations as basic instructions. The machine operators chosen contain all of the information required to perform one complete action required by the language.

The machine contains multiple parameter operators and a program counter that addresses the next instruction to be executed. Three registers are provided which hold eighteen digit numbers used for arithmetic operations along with a subscript stack that is used to compute subscript locations along with a set of flags that are used to pass branching information from one instruction to another.

Addresses in the machine are represented by 16 bit values. Any memory address greater than 20 hexadecimal is valid. Addresses less than 20 hexadecimal will be interpreted as having special significance. For example, addresses one through eight are reserved for subscript stack references. All other addresses in the machine are absolute addresses.

The arithmetic registers allow for the manipulation of signed numbers of up to eighteen decimal digits in length. Included in their representation is a sign indicator and the position of the assumed decimal point for the currently loaded number. While the form of the representation is not specified in the HYPO-COBOL document, it is necessary that there be no loss of precision for operations on numbers hav-

ing a full eighteen digits of significance.

There are two major types of numbers defined in the machine. The first is numbers in the DISPLAY mode. These numbers are represented in memory in the standard information exchange code for the peripherals. For microcomputers, the common representation would be in ASCII characters. These numbers may have separate signs indicated by "+" and "-" or may have a "zone" indicator added, denoting a negative sign. Packed decimal format is also available with numbers carried as sequential digit pairs stored in memory. The sign is indicated in the right-most position.

The following flags exist in the machine and can be checked by the instructions for a true or false value: BRANCH flag -- indicates if a branch is to be taken; END OF RECORD flag -- indicates that an end of input condition has been reached when an attempt was made to read input; OVERFLOW flag -- indicates the loss of information from a register due to a number exceeding the available size; INVALID flag -- indicates an invalid action in writing to a direct access storage device.

The following resources are required for a minimal implementation of this machine: a system input device capable of receiving low volume input, a system output device capable of displaying low volume output, and a direct access storage device capable of storing, reading, and writing files and programs.

B. MEMORY ORGANIZATION

Memory is divided into three major sections: (1) the data areas defined by the DATA DIVISION statements, (2) the code area, (3) and the constants area. No particular order of these sections is required. The first two areas assume the ability to both read and write, but the third only requires the ability to be read.

The data area contains variables defined by the DATA DIVISION statements, constants set in the WORKING STORAGE SECTION, and all file control blocks and buffers. These elements will be manipulated by the machine in accordance with the code instructions.

C. MACHINE OPERATIONS

1. Format

All of the machine operations consist of an operation number followed by a list of parameters. The sections that follow describe the various instructions, list the required parameters, and describe the actions taken by the machine in executing each instruction. As each instruction is fetched from memory, the program counter automatically increments by one.

2. Arithmetic operations

There are five arithmetic instructions which act only on the registers. In all cases, the result is placed

in register two. Operations are allowed to destroy the input values during the process of creating a result. Therefore, a number loaded into a register will not be available for a subsequent operation.

ADD: (addition). Sum the contents of register zero and register one.

Parameters: no parameters are required.

SUB: (subtract). Subtract register one from register zero.

Parameters: no parameters are required.

MUL: (multiply). Multiply register zero by register one.

Parameters: no parameters are required.

DIV: (divide). Divide register zero by the value in register one. The remainder is not retained.

Parameters: no parameters are required

RND: (round). Round register two to the last significant decimal place.

Parameters: no parameters are required.

3. Branching

All of the branching instructions are accomplished by changing the value of the program counter. Some are absolute branches and some test for condition flags that are set by the other instructions. Branches may also test the

state of the registers or perform direct comparisons on memory fields.

Several instructions use the same conditional branching conventions. First, the branch flag is checked for its current setting. If it is true, then a branch is made by changing the program counter to the value of the <branch address>. The branch flag is then set to false. If the flag was originally false, the program counter is incremented to the next sequential instruction.

BRN: (branch to an address). Load the program counter with the <branch address>.

Parameters: <branch address>

The next three instructions share a common format. The memory field addressed by the <memory address> is checked for the <address length>, and if all the characters match the test condition, then the branch flag is complemented. A conditional branch is taken after the test.

Parameters: <memory address> <address length> <branch address>

CAL: (compare alphabetic). Compare a memory field for alphabetic characters.

CNS: (compare numeric signed). Compare a field for numeric characters allowing for a sign character.

CNU: (compare numeric unsigned). Compare a field for numeric characters only.

D&C: (decrement a count and branch if zero). Decrement the value of the <address counter> by one, and if the result is zero, the program counter is set to the address given. If the result is not zero, then the program counter is incremented by four. If the result is zero before decrementing, the branch is taken.

Parameters: <address counter> <branch address>

EUR: (branch on end of records flag). If the end-of-records flag is true, it is set to false and the program counter is set to the <branch address>. If false, the program counter is incremented by two.

Parameters: <branch address>

GDP: (go to - depending on). The memory location addressed by the <number address> is read for the number of bytes indicated by the <memory length>. This number indicates which of the <branch addresses> is to be used. The first parameter is a bound on the number of branch addresses. If the number is within the range, the program counter is set to the indicated address. An out of bounds value causes the program counter to be advanced to the next sequential instruction.

Parameters: <bound number - byte> <memory length> <memory address> <branch addr-1> <branch addr-2> ... <branch addr-n>

INV: (branch if invalid-file-action flag true). If the invalid-file-action flag is true, then it is set to false, and the program counter is set to the branch ad-

dress. If it is false, the program counter is incremented by two.

Parameters: <branch address>

PER: (perform). The code address pointed to by the <change address> is loaded with the value of the <return address>. The program counter is then set to the <branch address>.

Parameters: <branch address> <change address> <return address>

RET: (return). If the value of the <branch address> is not zero, then the program counter is set to its value, and the <branch address> is set to zero. If the <branch address> is zero, the program counter is incremented by two.

Parameters: <branch address>

REQ: (register equal). This instruction checks for a zero value in register two. If it is zero, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RGT: (register greater than). Register two is checked for a negative sign. If present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

RLI: (register less than). Register two is checked for a positive sign, and if present, the branch flag is complemented. A conditional branch is taken.

Parameters: <branch address>

SEB: (branch on size error). If the overflow flag is true, then the program counter is set to the branch address, and the overflow flag is set to false. If it is false, then the program counter is incremented by two.

Parameters: <branch address>

The next three instructions all perform the same function and have the same general format. They compare two strings and perform a conditional branch. If the test condition is true, the branch flag is complemented prior to taking the conditional branch.

Parameters: <string addr-1> <string addr-2> <length - address> <branch address>

SEQ: (strings equal). Compare two string for equal characters.

SGT: (string greater than). Compare string one for greater than string two.

SLT: (string less than). Compare string one for less than string two.

4. Moves

The machine supports a variety of move operations for various formats and types of data. It does not support direct moves of numeric data from one memory field to another. Instead, all of the numeric moves go through the regis-

ters. This greatly reduced the number of instructions since all of the numeric types need to be supported by moves into and out of the registers for arithmetic operations.

The next seven instructions all perform the same function. They load a register with a numeric value and differ only in the type of number that they expect to see in memory at the <number address>. All seven cause the program counter to be incremented by five. Their common format is given below.

Parameters: <number address> <byte length> <byte decimal count> <byte register to load>

L0D: (load a numeric literal). Note that the decimal point indicator is not set in this instruction format. The literal will have an actual decimal point in it if required.

LD1: (load a numeric field).

LD2: (load a numeric field with an internal trailing sign).

LD3: (load a numeric field with an internal leading sign).

LD4: (load a numeric field with a separate leading sign).

LD5: (load a numeric field with a separate trailing sign).

LD6: (load a packed numeric field).

MED: (move into a alphanumeric edited field). The edit mask is loaded into the <to address> to set up the move, and then the <from address> information is loaded. The program counter is incremented by ten.

Parameters: <to address> <from address> <length of move>
<edit mask address> <edit mask length>

MNE: (move into a numeric edited field). First the edit mask is loaded into the receiving field, and then the information is loaded. Any decimal point alignment required will be performed. If truncation of significant digits is a side effect, the overflow flag is not set. The program counter is incremented by twelve.

Parameters: <to address> <from address> <address length of move> <edit mask address> <address mask length> <byte to decimal count> <byte from decimal count>

MOV: (move into an alphanumeric field). The memory field given by the <to address> is filled by the from field for the <move length> and then filled with blanks in the following positions for the <fill count>.

Parameters: <to address> <from address> <address move length> <address fill count>

SII: (store immediate register two). The contents of register two are stored into register zero and the decimal count and sign are indicators set.

Parameters: none.

The store instructions are grouped in the same order as the load instructions. Register two is stored into memory at the indicated location. Any alignment is performed, and if a non-zero leading digit is truncated by the operation, the overflow flag is set. All five of the store instructions cause the program counter to be incremented by four. The format for these instructions is as follows.

Parameters: <address to store into> <byte length> <byte decimal count>

SI0: (store into a numeric field).

SI1: (store into a numeric field with an internal trailing sign).

SI2: (store into a numeric field with an internal leading sign).

SI3: (store into a numeric field with a separate trailing sign).

SI4: (store into a numeric field with a separate leading sign).

SI5: (store into a packed numeric field).

5. Input-output

The following instructions perform input and output operations. The required operations are specified in the HYPO-COBOL manual, but the exact definitions of file formats and access methods are not defined. Files in this machine

are defined as having the following characteristics: they are either sequential or random, and, in general, files created in one mode are not required to be readable in the other mode. Standard files consist of fixed length records, and variable length files need not be readable in a random mode. Further, there must be some character or character string that delimits a variable length record.

ACC: (accept). Read from the system input device into memory at the location given by the <memory address>. The program counter is incremented by three.

Parameters: <memory address> <byte length of read>

CLS: (close). Close the file whose file control block is addressed by the <fcb address>. The program counter is incremented by two.

Parameters: <fcb address>

DIS: (display). Print the contents of the data field pointed to by <memory address> on the system output device for the indicated length. The program counter is incremented by three.

Parameters: <memory address> <byte length>

There are three open instructions with the same format. In each case, the file defined by the file control block referenced will be opened for the mode indicated. The program counter is incremented by two.

Parameters: <fcb address>

OPN: (open a file for input).

OP1: (open a file for output).

OP2: (open a file for both input and output). This is only valid for files on a random access device.

The following file actions all share the same format. Each performs a file action on the file referenced by the file control block. The record to be acted upon is given by the <record address>. The program counter is incremented by six.

Parameters: <fcb address> <record address> <record length - address>

DLS: (delete a record from a sequential file). Remove the record that was just read from the file. The file is required to be open in the input-output mode.

RDF: (read a sequential file). Read the next record into the memory area.

WIF: (write a record to a sequential file). Append a new record to the file.

RVL: (read a variable length record).

WVL: (write a variable length record).

RWS: (rewrite sequential). The rewrite operation writes a record from memory to the file, overlaying the last record that was read from the device. The file must be open



in the input-output mode.

The following file actions require random files rather than sequential files. They all make use of a random file pointer which consists of a <relative address> and a <relative length>. The memory field holds the number to be used in disk operations or contains the relative record number of the last disk action. The relative record number is the record count on the file starting with one. After the file action, the program counter is incremented by nine.

Parameters: <fcb address> <record address> <record length - address> <relative address> <relative length - byte>.

DLR: (delete a random record). Delete the record addressed by the relative record number.

RRR: (read random relative). Read a random record relative to the record number.

RRS: (read random sequential). Read the next sequential record from a random file. The relative record number of the record read is loaded into the memory reference.

RWR: (rewrite a random record).

WRR: (write random relative). Write a record into the area indicated by the memory reference.

WRS: (write random sequential). Write the next sequential record to a random file. The relative record

number is returned.

6. Special instructions

The remaining instructions perform special functions required by the machine that do not relate to any of the previous groups.

NOT: (negative test). Negate the value of the branch flag.

Parameters: no parameters are required.

LDI: (load a code address direct). Load the <code address> with the number indicated by the <memory address>.

Parameters: <code address> <memory address> <length - byte>

SCR: (calculate a subscript). Load the subscript stack with the value indicated from memory. The address loaded into the stack is the <initial address> plus an offset. Multiplying the <field length> by the number in the <memory reference> gives the offset value.

Parameters: <initial address> <field length> <memory reference> <memory length> <stack level>

SID: (stop with display). Display the indicated information and then stop.

Parameters: <memory address> <length - byte>

SIP: (stop). terminate the actions of the machine.

Parameters: no parameters are required.

The following instructions are used in setting up the machine environment and cannot be used in the normal execution of the machine.

BST: (backstuff). Resolve a reference to a label. Labels may be referenced prior to their definition, requiring a chain of resolution addresses to be maintained in the code. The latest location to be resolved is maintained in the symbol table and a pointer at that location indicates the next previous change. A zero pointer indicates no prior occurrences of the label. The code address referenced by <change address> is examined and if it contains zero, it is loaded with the <new address>. If it is not zero, then the contents are saved, and the process is repeated with the saved value as the change address after loading the <new address>.

Parameters: <change address> <new address>

INT: (initialize memory). Load memory with the <input string> for the given length at the <memory address>.

Parameters: <memory address> <address length> <input string>

SCD: (start code). Set the initial value of the program counter.

Parameters: <start address>

TER: (terminate). Terminate the initialization process and start executing code.

Parameters: no parameters are required.

III. MICRO-COBOL IMPLEMENTATION

A. COMPILER IMPLEMENTATION

1. General method

The LALR parser-table construction programs used here are based on the work of Knuth [9]. His work defines two methods of testing a grammar to see if it is LR(k). One of these methods leads to the creation of a set of tables that can be used to drive the parse actions of a compiler. While difficult to implement in the form given by Knuth, the method has been developed in usable form for subsets of the grammars that are LR(k). References 2 and 3 contain detailed discussions of the methods currently available. The algorithm used to develop the tables for the MICRO-COBOL compiler was developed by W. Lalonde [12].

The compiler was designed to read the source language statements from a diskette or other mass storage device, extract the needed information for the symbol table, and write the output code back onto the diskette all in one pass of the source program. The grammar was initially defined for the entire language, but the size constraints placed on the implementation required smaller tables. The grammar was then defined in two parts which run in succession. The major method of passing information from the

first part to the second is by placing the information in the symbol table.

The output code from the compiler consists of the operations that have been previously defined. They were designed as an intermediate language that would be executed by the interpreter described in section B. The vast differences between the operations available for the target computer and the operations necessary to support COBOL made this approach easier than 8080 machine code.

2. Control flow

The compiler has been designed so that the operation of the two parts would be transparent to the user. When the first part is loaded it brings in with its code a reader program which loads the second file automatically. Prior to calling the reader program, the first part writes any pending code to the disk and loads all toggles to a common area ready to be read by the second part.

Internally, the control of the two parts is identical. The parser is called after initialization and runs until it either finishes its task or reaches an unrecoverable error state. The major subroutines in the compiler are the scanner and the production case statement. Both are controlled in their actions by the parser.

3. Internal structures

The major internal structure is the symbol table. It was designed as a list where the elements in the list are the descriptions of the various symbols in the program. As new symbols are encountered they are added to the end of the list. Symbols already in the list can be accessed through the use of a "current symbol pointer." The location of items in the list is determined by checking the identifier against a hash table that points to the first entry in the symbol table with that hash code. A chain of collision addresses is maintained in the symbol table which links entries which have the same hash value.

All of the items in the symbol table contain the following information: a collision field, a type field, the length of the identifier, and the address of the item. If an item in the symbol table is a data field, the following information is included in the table: the length of the item, the level of the data field, an optional decimal count, an optional multiple occurrence count, and the address of the edit field, if required. If the item is a file name then the following additional information is included: the file record length, the file control block address, and the optional symbol table location of the relative record pointer. If the item is a label, then the only additional information is the location of the return instruction at the end of the paragraph or section.

In addition to the symbol table, two stacks are used for storing information: the level stack and the identifier stack. In both cases, they are used to hold pointers to entries in the symbol table. The identifier stack is used to collect multiple occurrences in such statements as the GO TO - DEPENDING statement. The level stack is used to hold information about the various levels that make up a record description.

The parser has control of a set of stacks that are used in the manipulation of the parse states. In addition to the state stack that is required by the parser, part one has a value stack and part two has two different value stacks that operate in parallel with the parser state stack. The use of these stacks is described below.

4. Part one

The first part of the compiler is primarily concerned with building the symbol table that will be used by the second part. The actions corresponding to each parse step are explained in the sections that follow. In each case, the grammar rule that is being applied is given, and an explanation of what program actions take place for that step has been included. In describing the actions taken for each parse step there has been no attempt to describe how the symbol table is constructed or how the values are preserved on the stack. The intent of this section is to describe what information needs to be retained and at what

point in the parse it can be determined. Where no action is required for a given statement, or where the only action is to save the contents of the top of the stack, no explanation is given. Questions regarding the actual manipulation of information should be resolved by consulting the programs.

- 1 <program> ::= <id-div> <e-div> <d-div> PROCEDURE
Reading the word PROCEDURE terminates the first part of the compiler.
- 2 <id-div> ::= IDENTIFICATION DIVISION. PROGRAM-ID.
 <comment> . <auth> <date> <sec>
- 3 <auth> ::= AUTHOR . <comment> .
- 4 | <empty>
- 5 <date> ::= DATE-WRITTEN . <comment> .
- 6 | <empty>
- 7 <sec> ::= SECURITY . <comment> .
- 8 | <empty>
- 9 <comment> ::= <input>
- 10 | <comment> <input>
- 11 <e-div> ::= ENVIRONMENT DIVISION . CONFIGURATION SECTION.
 <src-obj> <i-o>
- 12 <src-obj> ::= SOURCE-COMPUTER . <comment> <debug> .
 OBJECT-COMPUTER . <comment> .
- 13 <debug> ::= DEBUGGING MODE
Set a scanner toggle so that debug lines will be read.
- 14 | <empty>
- 15 <i-o> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .


```

        <file-control-list> <ic>
16         ! <empty>
17 <file-control-list> ::= <file-control-entry>
18         ! <file-control-list> <file-control-entry>
19 <file-control-entry> ::= SELECT <id> <attribute-list> .

```

At this point all of the information about the file has been collected and the type of the file can be determined. File attributes are checked for compatibility and entered in the symbol table.

```

20 <attribute-list> ::= <one attrib>
21         ! <attribute-list> <one attrib>
22 <one-attrib> ::= ORGANIZATION <org-type>
23         ! ACCESS <acc-type> <relative>
24         ! ASSIGN <input>

```

A file control block is built for the file using an INT operator.

```

25 <org-type> ::= SEQUENTIAL

```

No information needs to be stored since the default file organization is sequential.

```

26         ! RELATIVE

```

The relative attribute is saved for production 19.

```

27 <acc-type> ::= SEQUENTIAL

```

This is the default.

```

28         ! RANDOM

```

The random access mode needs to be saved for production 19.

```

29 <relative> ::= RELATIVE <id>

```

The pointer to the identifier will be retained by the

current symbol pointer, so this production only saves a flag on the stack indicating that the production did occur.

```
30          ! <empty>
31 <ic> ::= I-O-CONTROL . <same-list>
32          ! <empty>
33 <same-list> ::= <same-element>
34          ! <same-list> <same-element>
35 <same-element> ::= SAME <id-string> .
36 <id-string> ::= <id>
37          ! <id-string> <id>
38 <d-div> ::= DATA DIVISION . <file-section> <work> <link>
39 <file-section> ::= FILE SECTION . <file-list>
```

Actions will differ in production 64 depending upon whether this production has been completed. A flag needs to be set to indicate completion of the file section.

```
40          ! <empty>

The flag, indicated in production 39, is set.
41 <file-list> ::= <file-element>
42          ! <file-list> <file-element>
43 <files> ::= FD <ic> <file-control> . <record-description>
```

This statement indicates the end of a record description, and the length of the record and its address can now be loaded into the symbol table for the file name.

```
44 <file-control> ::= <file-list>
45          ! <empty>
```



```

46 <file-list> ::= <file-element>
47           ! <file-list> <file-element>
48 <file-element> ::= BLOCK <integer> RECORDS
49           ! RECORD <rec-count>

```

The record length can be saved for comparison with the calculated length from the picture clauses.

```

50           ! LABEL RECORDS STANDARD
51           ! LABEL RECORDS OMITTED
52           ! VALUE OF <id-string>
53 <rec-count> ::= <integer>
54           ! <integer> TO <integer>

```

The TO option is the only indication that the file will be variable length. The maximum length must be saved.

```

55 <work> ::= WORKING-STORAGE SECTION . <record-description>
56           ! <empty>
57 <link> ::= LINKAGE SECTION . <record-description>
58           ! <empty>
59 <record-description> ::= <level-entry>
60           ! <record-description> <level-entry>
61 <level-entry> ::= <integer> <data-id> <redefines>
               <data-type> .

```

The level entry needs to be loaded into the level stack. The level stack is used to keep track of the nesting of field definitions in a record. At this time there may be no information about the length of the item being defined, and its attributes may depend entirely upon its constituent fields. If there is a

pending literal, the stack level to which it applies is saved.

62 <data-id> ::= <id>

63 ; FILLER

An entry is built in the symbol table to record information about this record field. It cannot be used explicitly in a program because it has no name, but its attributes will need to be stored as part of the total record.

64 <redefines> ::= REDEFINES <id>

The redefines option gives new attributes to a previously defined record area. The symbol table pointer to the area being redefined is saved so that information can be transferred from one entry to the other. In addition to the information saved relative to the redefinition, it is necessary to check to see if the current level number is less than or equal to the level recorded on the top of the level stack. If this is true, then all information for the item on the top of the stack has been saved and the stack can be reduced.

65 ; <empty>

As in production 64, the stack is checked to see if the current level number indicates a reduction of the level stack. In addition, special action needs to be taken if the new level is 01. If an 01 level is encountered at this production prior to production 39 or 40 (the end of the file area), it is an implied rede-

inition of the previous 01 level. In the working storage section, it indicates the start of a new record.

66 <data-type> ::= <prop-list>

67 ! <empty>

68 <prop-list> ::= <data-element>

69 ! <prop-list> <data-element>

70 <data-element> ::= PIC <input>

The <input> at this point is the character string that defines the record field. It is analyzed and the extracted information is stored in the symbol table.

71 ! USAGE COMP

The field is defined to be a packed numeric field.

72 ! USAGE DISPLAY

The DISPLAY format is the default, and thus no special action occurs.

73 ! SIGN LEADING <separate>

This production indicates the presence of a sign in a numeric field. The sign will be in a leading position. If the <separate> indicator is true, then the length will be one longer than the picture clause, and the type will be changed.

74 ! SIGN TRAILING <separate>

The same information required by production 73 must be recorded, but in this case the sign is trailing rather than leading.

75 ! OCCURS <integer>

The type must be set to indicate multiple occurrences,

and the number of occurrences saved for computing the space defined by this field.

76 ! SYNC <direction>

Synchronization with a natural boundary is not required by this machine.

77 ! VALUE <literal>

The field being defined will be assigned an initial value determined by the value of the literal through the use of an INT operator. This is only valid in the WORKING-STORAGE SECTION.

78 <direction> ::= LEFT

79 ; RIGHT

80 ; <empty>

81 <separate> ::= SEPARATE

The separate sign indicator is set on.

82 ! <empty>

83 <literal> ::= <input>

The input string is checked to see if it is a valid numeric literal, and if valid, it is stored to be used in a value assignment.

84 ! <lit>

This literal is a quoted string.

85 ! ZERO

As is the case of all literals, the fact that there is a pending literal needs to be saved. In this case and the three following cases, an indicator of which literal constant is being saved is all that is required. The literal value can be reconstructed

later.

86 ! SPACE

87 ! QUOTE

88 <integer> ::= <input>

The input string is converted to an integer value for later internal use.

89 <id> ::= <input>

The input string is the name of an identifier and is checked against the symbol table. If it is in the symbol table, then a pointer to the entry is saved. If it is not in the symbol table, then an entry is added and the address of that entry is saved.

5. Part two

The second part includes all of the PROCEDURE DIVISION, and is the part where code generation takes place. As in the case of the first part, there was no intent to show how various pieces of information were retrieved but only what information was used in producing the output code.

1 <p-div> ::= PROCEDURE DIVISION <using> .

 <proc-body> END .

This production indicates termination of the compilation. If the program has sections, then it will be necessary to terminate the last section with a REEL 0 instruction. The code will be ended by the output of a T&R operation.

2 <using> ::= USING <id-string>

3 ! <empty>

4 <id-string> ::= <id>

The identifier stack is cleared and the symbol table address of the identifier is loaded into the first stack location.

5 ! <id-string> <id>

The identifier stack is incremented and the symbol table pointer stacked.

6 <proc-body> ::= <paragraph>

7 ! <proc-body> <paragraph>

8 <paragraph> ::= <id> . <sentence-list>

The starting and ending address of the paragraph are entered into the symbol table. A return is emitted as the last instruction in the paragraph (RET 0). When the label is resolved, it may be necessary to produce a BST operation to resolve previous references to the label.

9 ! <id> SECTION .

The starting address for the section is saved. If it is not the first section, then the previous section ending address is loaded and a return (RET 0) is output. As in production 8, a BST may be produced.

10 <sentence-list> ::= <sentence>

11 ! <sentence-list> <sentence> .

12 <sentence> ::= <imperative>

13 ! <conditional>

14 ! ENTER <id> <opt-id>

This construct is not implemented. An ENTER allows

statements from another language to inserted in the source code.

15 <imperative> ::= ACCEPT <subid>

ACC <address> <length>

16 ! <arithmetic>

17 ! CALL <lit> <using>

This is not implemented.

18 ! CLOSE <id>

CLS <file control block address>

19 ! <file-act>

20 ! DISPLAY <lit/id> <opt-lit/id>

The display operator is produced for the first literal or identifier (DIS <address> <length>). If the second value exists, the same code is also produced for it.

21 ! EXIT <program-id>

RET 0

22 ! GO <id>

BRN <address>

23 ! GO <id-string> DEPENDING <id>

GDP is output, followed by a number of parameters: <the number of entries in the identifier stack> <the length of the depending identifier> <the address of the depending identifier> <the address of each identifier in the stack>.

24 ! MOVE <lit-id> TO <subid>

The types of the two fields determine the move that is generated. Numeric moves go through register two using a load and a store. Non-numeric moves depend upon

the result field and may be either MOV, MED or MNE. Since all of these instructions have long parameter lists, they have not been listed in detail.

25 ! OPEN <type-action> <id>

This produces either OPN, OP1, or OP2 depending upon the <type-action>. Each of these is followed by a file control block address.

26 ! PERFORM <id> <thru> <finish>

The PER operation is generated followed by the <branch address> <the address of the return statement to be set> and <the next instruction address>.

27 ! <read-id>

28 ! STOP <terminate>

If there is a terminate message, then SPD is produced followed by <message address> <message length>. Otherwise STP is emitted.

29 <conditional> ::= <arithmetic> <size-error> <imperative>

A BST operator is output to complete the branch around the imperative from production 65.

30 ! <file-act> <invalid> <imperative>

A BST operator is output to complete the branch from production 64.

31 ! IF <condition> <action> ELSE <imperative>

Two BST operators are required. The first fills in the branch to the ELSE action. The second completes the branch around the <imperative>.

32 ! <read-id> <special> <imperative>

A BST is produced to complete the branch around the

<imperative>.

33 <Arithmetic> ::= ADD <l/id> <opt-l/id> TO <subid> <round>

The existence of multiple load and store instructions make it difficult to indicate exactly what code will be generated for any of the arithmetic instructions. The type of load and store will depend on the nature of the number involved, and in each case the standard parameters will be produced. This parse step will involve the following actions: first, a load will be emitted for the first number into register zero. If there is a second number, then a load into register one will be produced for it, followed by an ADD and a SII. Next a load into register one will be generated for the result number. Then an ADD instruction will be emitted. Finally, if the round indicator is set, a RND operator will be produced prior to the store.

34 ; DIVIDE <l/id> INTO <subid> <round>

The first number is loaded into register zero. The second operand is loaded into register one. A DIV operator is produced, followed by a RND operator prior to the store, if required.

35 ; MULTIPLY <l/id> BY <subid> <round>

The multiply is the same as the divide except that a MUL is produced.

36 ; SUBTRACT <l/id> <opt-l/id> FROM
 <subid> <round>

Subtraction generates the same code as the ADD except that a SUB is produced in place of the last ADD.

37 <file-act> ::= DELETE <id>

Either a DLS or a DLR will be produced along with the required parameters.

38 ! REWRITE <id>

Either a RWS or a RWR is emitted, followed by parameters.

39 ! WRITE <id> <special-act>

There are four possible write instructions: WTF, WVL, WRS, and WRR.

40 <condition> ::= <lit/id> <not> <cond-type>

One of the compare instructions is produced. They are CAL, CNS, CNU, RGT, RLT, REQ, SGT, SLT, and SEQ. Two load instructions and a SUB will also be emitted if one of the register comparisons is required.

41 <cond-type> ::= NUMERIC

42 ! ALPHABETIC

43 ! <compare> <lit/id>

44 <not> ::= NOT

NEG

45 ! <empty>

46 <compare> ::= GREATER

47 ! LESS

48 ! EQUAL

49 <ROUND> ::= ROUNDED

50 ! <empty>

51 <terminate> ::= <literal>

52 ! RUN

53 <special> ::= <invalid>

An ERO operator is produced followed by a zero. The zero acts as a filler in the code and will be back-stuffed with a branch address. In this production and several of the following, there is a forward branch on a false condition past an imperative action. For an example of the resolution, examine production 32.

55 <opt-id> ::= <subid>

56 ; <empty>

57 <action> ::= <imperative>

BRN 0

58 ; NEXT SENTENCE

BRN 0

59 <thru> ::= THRU <id>

60 ; <empty>

61 <finish> ::= <l/id> TIMES

LDI <address> <length> DEC 0

62 ; UNTIL <condition>

63 ; <empty>

64 <invalid> ::= INVALID

INV 0

65 <size-error> ::= SIZE ERROR

SER 0

66 <special-act> ::= <when> ADVANCING <how-many>

67 ; <empty>

68 <when> ::= BEFORE

69 ; AFTER

70 <how-many> ::= <inteder>


```

71          | PAGE
72 <type-action> ::= INPUT
73          | OUTPUT
74          | I-O

```

```

75 <subid> ::= <subscript>
76          | <id>

```

```

77 <integer> ::= <input>

```

The value of the input string is saved as an internal number. •

```

78 <id> ::= <input>

```

The identifier is checked against the symbol table, if it is not present, it is entered as an unresolved label.

```

79 <l/id> ::= <input>

```

The input value may be a numeric literal. If so, it is placed in the constant area with an INI operand. If it is not a numeric literal, then it must be an identifier, and it is located in the symbol table.

```

80          | <subscript>

```

```

81          | ZERO

```

```

82 <subscript> ::= <id> ( <input> )

```

If the identifier was defined with a USING option, then the input string is checked to see if it is a number or an identifier. If it is an identifier, then an SCR operator is produced.

```

83 <opt-l/id> ::= <l/id>

```

```

84          | <empty>

```

```

85 <nn-lit> ::= <lit>

```


The literal string is placed into the constant area using an INT operator.

```
86          ! SPACE
87          ! QUOTE
88 <literal> ::= <nn-lit>
89          ! <input>
```

The input value must be a numeric literal to be valid and is loaded into the constant area using an INT.

```
90          ! ZERO
91 <lit/id> ::= <l/id>
92          ! <nn-lit>
93 <opt-lit/id> ::= <lit/id>
94          ! <empty>
95 <program-id> ::= <id>
96          ! <empty>
97 <read-id> ::= READ <id>
```

There are four read operations: RDF, RVL, RRS, and RRR.

The output code file is the only product of the compiler that is retained. All of the needed information has been extracted from the symbol table, and it is not required by the interpreter. Code will be generated for all programs including those that contain errors and can be examined through the use of the decode program. This program translates the output file into a listing of code operators followed by the parameters.

B. INTERPRETER IMPLEMENTATION

1. General structure

The format that has been presented for the output code determines the general form of the interpreter. If it had not been possible to transform the instructions from the compiler into a set of call-like commands, it would have been necessary to implement a stack in the interpreter. In general, the interpreter contains a large "case statement" which decodes each operation and either calls subroutines to perform the required actions or acts directly on the run-time environment to control the actions of the interpreter. All communication between instructions is done through common areas in the program where information can be stored for later use.

The design of the interpreter has been modularized in an attempt to allow easy transition to other hardware configurations and operating systems. If desired, any section of the instructions could be implemented in assembly language modules or could be passed to the operating system for action. The entire system has been coded in PL/M for consistency, ease of development, and maximum portability [7].

2. Code modules

The following sections explain the interpreter by noting the specific manner in which the machine instructions

defined in section II-C have been implemented. The divisions are the same as those in section II-C.

a. Arithmetic instructions

Since the machine was defined as having only one set of arithmetic registers, it was necessary to convert all numeric input to one form. The packed decimal format was chosen as the format that would be used in the registers. This conversion process slows down the arithmetic operations slightly, but the reduction of the interpreter memory size was considered more important.

All of the arithmetic operations take place in a set of three work areas or registers. Each of these areas is ten bytes long and can contain an eighteen digit number with one fill character on each end. The extra space facilitates checking for overflow and also makes rounding operations easier. The language does not support the COMPUTE verb, so no storage of intermediate results is required from one instruction to another.

All of the arithmetic instructions use the packed decimal feature of the 8080 as a basis for their actions. Each of the instructions depends on the basic operation of adding two registers: subtraction is accomplished using nines complement arithmetic, multiplication is done through a shift and add algorithm, and division by a shift and subtract method.

If the amount of computations required by a given application make it necessary to speed up these instructions, they could be replaced by a package in assembly language. Extending the grammar to include the COMPUTE verb would require changes in the compiler to allow for temporary locations, but it could be included.

b. Branching

The operation of the interpreter is controlled by a program counter that points to the next operation to be performed. All branching is done by changing the normal sequential order of execution of instructions. In addition to acting directly on the program counter, branching instructions use the branch flag to determine when changes should be made. All of the addresses that point to code are absolute addresses and can be loaded directly into the program counter.

c. Input-output operations

All of the input and output operations use the CP/M interface capabilities [5]. The program expects to see the files in the form that the CP/M editor would have created them. The physical records on the disk are assumed to be 128 bytes in length and have all logical records ending with a carriage-return and a line-feed sequence. There is only one type of file under CP/M, so all restrictions on mixing modes of files are removed for fixed length files. Files created in one program as sequential can be accessed as ran-

dom files in another program. Variable length files cannot be accessed in a random fashion because there is no way to compute the starting address of each record.

Where possible, the interface routines have been localized in the programs to simplify transportation to another operating environment. Items relating to file control blocks, disk record lengths, and other system parameters have been established as literals in the programs, rather than entered as numbers, so that changes will not have to be made throughout the code.

a. Moves

As noted previously, the machine lacks numeric moves. There were two major reasons for leaving out the various moves of numeric data. The first was that the added moves would have required more program space, and the second was to simplify the coding and checking of the program. Since all of the numeric types are supported with register load and store operations, any move can be accomplished by a load into register two and a store into the result field.

Alpha-numeric moves are supported as direct moves from memory to memory. If speed is required for a numeric move, the fields concerned can be redefined as alpha-numeric and the memory move used. However, this type of move will only work on two numbers that have exactly the same representation in the computer.

Edited moves also are from memory to memory, but they involve several additional steps. The edit mask is loaded into the result field before any characters are loaded, and each character in both the receiving field and the sending field is examined to determine what action should be taken in addition to a move.

3. Limitations

The MICRO-COBOL implementation did not lend itself to support of the Interprogram Communications Module. There was no capability in the operating system to dump the memory image onto the disk or to restore it. It would be possible to implement such a supervisor call, or a one way call could perhaps be implemented from one program to another without the possibility of a return to the calling program. If required by an application where modification of the operating system was not practical, a small overlay program could be written as an independent function to be loaded with the interpreter. If large systems are to be run on microcomputers with minimal memory, some type of interprogram communications would greatly facilitate their design.

C. SOFTWARE TOOLS

As in any software development, one of the things that was most important to the success of this project was the software support for the development effort. This system was developed on the 360/67 rather than on the 8080. Using

the Intel INTERP program [8] and the CP/M simulator developed by at the Naval Postgraduate School [11], it was possible to both compile programs on CP/CMS and run the generated code. This facility removed the necessity of transporting code from the 360 to the 8080 for testing and greatly improved the productivity.

Using the simulator did not result in exactly the same product as would have been developed if the project had been done entirely on the 8080. It was not possible to load a program on the simulator without destroying the core image currently in the simulator. In particular, the first part of the compiler could not leave the symbol table for the second part if the second part was loaded by a normal load. This problem was resolved by writing a set of small programs that read in the sequence of compiler components from simulated memory image files. These programs have been included in this document so that, if future work is done, the simulator could be used again.

IV. CONCLUSIONS

This project demonstrates the feasibility of applying modern compiler construction techniques to the implementation of a language developed prior to the work on formal grammars. Not only is it possible to construct a compiler for HYPO-COBOL using an LALR(1) parser, but the resulting programs are highly compact. This allows the implementation of the compiler on smaller machines and increases the number of target systems.

Only a limited number of programs have been written using the compiler, and no attempt has been made to train others in its use. However, adapting to the subset should not be a major problem for a programmer experienced in writing standard COBOL. There have been no extensive timing tests of the system, but current indications are that both the compiler and interpreter operate at an acceptable rate.

There are several areas that could be enhanced in this implementation of HYPO-COBOL. One of these areas is the interprogram communication module. Due to the limitations on core size usually imposed by microcomputer systems, it would be very helpful to be able to compile a set of programs that could be used together as a single module. Several ideas were presented in the body of this paper which indicate how the interprogram communication module could be developed.

The GIVING option for arithmetic statements could be added to the grammar. This option would improve computational programs, and could be supported without change to the existing interpreter. As discussed previously, the COMPUTE verb could be added if desired, but it would require greater changes both to the grammar and to the interpreter.

Programmers that have used COBOL in a standard implementation will find the appearance of the WORKING-STORAGE SECTION quite different due to the lack of the 77 level. No restriction was placed on the size of the level numbers other than they must be less than 255. This allows for the standard practice of level skipping. In addition, it would not be difficult to make the 77 level perform in a normal manner. There is no difference in the way that the language considers an 01 level and a 77 level item, but the compatibility with common usage would be very helpful to a COBOL programmer.

It is hoped that the results of this project are in a form that will allow others to use the compiler as a working system. It is recognized that many undiscovered problems will plague the initial users, but every effort has been made to describe what the system should do and to isolate the functions within the interpreter to facilitate changes.

APPENDIX A - MICRO-COBOL USERS MANUAL

This manual is written to explain the implementation of HYPO-COBOL done at the Naval Postgraduate School for the Intel 8080 microcomputer running with CP/M (Control Program / Microcomputer). It is not intended that this manual take the place of the HYPO-COBOL specification but that it supply information on the manner in which this implementation was done. There is no attempt to teach COBOL; however, someone who has a working knowledge of the language should be able to produce programs from the information contained in this manual.

This manual contains a brief overview of the justification for HYPO-COBOL and the organization of this implementation. It contains a brief explanation of each of the constructs available in the language and shows samples of their use. It explains the interactions between the various parts of the compiler and interpreter and how they interface with the operating system. It also includes a list of references that might be useful to someone who wished to modify the compiler.

One of the major goals of this document is to explain how the operating system used effects the operation of the compiler. It is recognized that if the implementation is to be useful it will need to be modified to run on other confi-

gurations of hardware and on other operating systems. Where it was possible, the interaction with the operating environment was insulated from the other parts of the program, but in the case of the file structure certain assumptions had to be made that could require modification.

ACKNOWLEDGEMENT

Any organization interested in reproducing the COBOL report and specifications in whole or in part, using ideas from this report as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication. Any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgement of the source, but need not quote the acknowledgment.

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations. No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), programming for the Univac (R) I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM commercial Translator Form No. F 28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

CONTENTS

I. HYP <u>U</u> -COBOL OVERVIEW.....	64
II. ORGANIZATION OF THE IMPLIMENTATION.....	66
III. MICRO-COBOL ELEMENTS.....	68
IDENTIFICATION DIVISION Format.....	70
ENVIRONMENT DIVISION Format.....	71
<file-control-entry>	72
DATA DIVISION Format.....	74
<comment>	76
<data-description-entry> Format.....	77
PROCEDURE DIVISION Format.....	79
<sentence>	80
<imperative-statement>	81
<conditional-statements>	82
ACCEPI	83
ADD	84
CALL	85
CLOSE	86

DELETE	87
DISPLAY	88
DIVIDE	89
ENTER	90
EXIT	91
GO	92
IF	93
MOVE	94
MULTIPLY	95
OPEN	96
PERFORM	97
READ	98
REWRITE	99
STOP	100
SUBTRACT	101
WRITE	102
<condition>	103
Subscripting	104
IV. COMPILER TOGGLES.....	105

V. RUN TIME CONVENTIONS.....	106
VI. FILE INTERACTIONS WITH CP/M.....	107
ERROR MESSAGES.....	109
LIST OF REFERENCES.....	114

I. HYPO-COBOL OVERVIEW

In order to provide a standard COBOL subset that could be implemented on a small computer system, the Department of the Navy has defined HYPO-COBOL. This definition is intended to give the minimum subset of the COBOL language that would be useable as a working product. This subset does not agree with the lowest level of COBOL as defined by the CODASYL group and in some cases includes only a portion of one of the COBOL levels as defined in the current standards. It is defined to include a portion of the NUCLEUS and both SEQUENTIAL I-O and RELATIVE I-O. A small portion of the DEBUG module was included along with some INTERPROGRAM COMMUNICATION instructions.

Where possible, short forms were included rather than long forms, and if two forms existed for the same instruction, only one was included. For example, the shortened PIC is used rather than the full word PICTURE. Also GO is not followed by the optional word IO. This does allow the definition to be a proper subset of the standard COBOL, but, at the same time, reduces the impact of the wordiness of COBOL on a small system.

As an exception to the general rule, PERFORM UNTIL was included from level 2 of the NUCLEUS in order to provide an additional control structure to support structured programming.

ming techniques. Further information on HYP0-C0B0L can be found in reference 6.

II. ORGANIZATION OF THE IMPLIMENTATION

The compiler is designed to run on an 8080 system in an interactive mode through the use of a teletype or console. It requires at least 12k of RAM memory and a mass storage device for reading and writing. The compiler is composed of two parts or passes, each of which reads a portion of the input file. Pass one reads the input program and builds the symbol table. At the end of the DATA DIVISION, pass one is overlayed by pass two which uses the symbol table to produce the code. The output code is written as it is produced to minimize the use of internal storage.

The first program of the interpreter builds the core image of the code and performs such functions as back-stuffing addresses. This first program loads the second program in and relenquishes control to the run time environment. The interpreter is controlled by a large case statement that decodes the instructions and performs the required actions.

As a tool for debugging the compiler a seperate program was created that will read the output code and translate the operations back into the mnemonics that are used in the second pass of the compiler. This "decode" program has been included with the other programs in order that anyone wishing to make chanches to the output code or to the actions of

the interpreter can use this tool.

III. MICRO-COBOL ELEMENTS

This section contains a description of each element in the language and shows simple examples of its use. The following conventions are used in explaining the formats: Elements inclosed in broken braces `< >` are themselves complete entities and are described elsewhere in the manual. Elements inclosed in stacks of braces `{ }` are choices, one of the elements which is to be used. Elements inclosed in brackets `[]` are optional. All elements in capital letters are reserved words and must be spelled exactly.

User names are indicated as lower case. These names have been restricted to 12 characters in length. There are no restrictions in the compiler on what characters may be in a user name. Some restrictions do need to be made to assure that they are not taken as literal numbers when used in the DATA DIVISION. For example a record could be defined in the DATA DIVISION with the name 1234, but the command `MOVE 1234 TO RECORD1` would result in the movement of the literal number not the data stored. The HYPO-COBOL description requires that each name start with a letter. This restriction was not implemented because it violates common programming practices.

The input to the compiler does not need to conform to standard COBOL format. Freeform input will be accepted as

the default condition. If desired, sequence numbers can be entered in the first six positions of each line. However, a toggle needs to be set to cause the combiler to ignore those lines.

ELEMENT:

IDENTIFICATION DIVISION Format

FORMAT:

IDENTIFICATION DIVISION.
PROGRAM-ID. <comment>.
[AUTHOR. <comment>.]
[DATE-WRITTEN. <comment>.]
[SECURITY. <comment>.]

DESCRIPTION:

This division provides information for program identification for the reader. The order of the lines is fixed.

EXAMPLES:

IDENTIFICATION DIVISION.
PROGRAM-ID. SAMPLE.
AUTHOR. A S CRAIG.

ELEMENT:

ENVIRONMENT DIVISION Format

FORMAT:

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. <comment> (DEBUGGING MODE).

OBJECT-COMPUTER. <comment>.

[INPUT-OUTPUT SECTION.

FILE-CONTROL.

<file-control-entry> . . .

[I-O-CONTROL.

SAME file-name-1 file-name-2 [file-name-3]

[file-name-4] [file-name-5].]]

DESCRIPTION:

This division determines the external nature of a file. In the case of CP/M all of the files used can be accessed either sequentially or randomly except for variable length files which are sequential only. The debugging mode is also set by this section.

ELEMENT:

<file-control-entry>

FORMAT:

1.

```
SELECT file-name
      ASSIGN implementor-name
      (ORGANIZATION SEQUENTIAL)
      (ACCESS SEQUENTIAL).
```

2.

```
SELECT file-name
      ASSIGN implementor-name
      ORGANIZATION RELATIVE
      (ACCESS {SEQUENTIAL [RELATIVE data-name]}).
              {RANDOM RELATIVE data-name }
```

DESCRIPTION:

The file-control-entry defines the type of file that the program expects to see. There is no difference on the diskette, but the type of reads and writes that are performed will differ. For CP/M the implementor name needs to conform to the normal specifications.

EXAMPLES:

SELECT CARDS

ASSIGN CARD.FIL.

SELECT RANDOM-FILE

ASSIGN A.RAN

ORGANIZATION RELATIVE

ACCESS RANDOM RELATIVE RAND-FLAG.

ELEMENT:

DATA DIVISION Format

FORMAT:

DATA DIVISION.

[FILE SECTION.

[FD file-name

IBLOCK integer-1 RECORDS]

[RECORD [integer-2 TO] integer-3]

 [LABEL RECORD {STANDARD}]
 {OMITTED }]

[VALUE OF implementor-name-1 literal-1

[implementor-name-2 literal-2] ...].

[<record-description-entry>] ...] ...

[WORKING-STORAGE SECTION.

[<record-description-entry>] ...]

[LINKAGE SECTION.

[<record-description-entry>] ...]

DESCRIPTION:

This is the section that describes how the data is structured. There are no major differences from standard COBOL except for the following: 1. Label records make no sense on the diskette so no entry is

required. 2. The VALUE OF clause likewise has no meaning for CP/M. 3. The linkage section has not been implemented.

If a record is given two lengths as in RECORD 12 10 128, the file is taken to be variable length and can only be accessed in the sequential mode. See the section on files for more information.

ELEMENT:

<comment>

FORMAT:

any string of characters

DESCRIPTION:

A comment is a string of characters. It may include anything other than a period followed by a blank or a reserved word, either of which terminate the string. Comments may be empty if desired, but the terminator is still required by the program.

EXAMPLES:

this is a comment
anotheroneallruntogether
8080b 16K

ELEMENT:

<data-description-entry> Format

FORMAT:

```

level-number {data-name}
              {FILLER }

[REDEFINES data-name]

[PIC character-string]

[USAGE {COMP  } ]
      {DISPLAY}

[SIGN {LEADING} [SEPARATE]]
      {TRAILING}

[OCCURS integer]

[SYNC {LEFT } ]
      {RIGHT}

[VALUE literal].

```

DESCRIPTION:

This statement describes the specific attributes of the data. Since the 8080 is a byte machine, there was no meaning to the SYNC clause, and thus it has not been implemented.

EXAMPLES:

01 CARD-RECORD.

02 PART PIC X(5).

02 NEXT-PART PIC 99V99 USAGE COMP.

02 FILLER.

03 NUMB PIC \$9(3)V9 SIGN LEADING SEPARATE.

03 LONG-NUMB 9(15).

03 STRING REDEFINES LONG-NUMB PIC X(15).

02 ARRAY PIC 99 OCCURS 100.

ELEMENT:

PROCEDURE DIVISION Format

FORMAT:

1.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].  
section-name SECTION.  
[paragraph-name. <sentence> [<sentence> ... ] ... ] ...
```

2.

```
PROCEDURE DIVISION [USING name1 [name2] ... [name5]].  
paragraph-name. <sentence> [<sentence> ...] ...
```

DESCRIPTION:

As is indicated, if the program is to contain sections, then the first paragraph must be in a section. The USING option is part of the interprogram communication module and has not been implemented.

ELEMENT:

<sentence>

FORMAL:

<imperative-statement>

<conditional-statement>

ENTER verb

DESCRIPTION:

All sentences other than ENTER fall in one of the two main categories. ENTER is part of the interprogram communication module.

ELEMENT:

<imperative-statement>

FORMAT:

The following verbs are always imperatives:

ACCEPT

CALL

CLOSE

DISPLAY

EXIT

GO

MOVE

OPEN

PERFORM

STOP

The following may be imperatives:

arithmetic verbs without the SIZE ERROR statement

and DELETE, WRITE, and REWRITE without the INVALID option.

ELEMENT:

<conditional-statements>

FORMAT:

IF

READ

.

arithmetic verbs with the SIZE ERROR statement

and DELETE, WRITE, and REWRITE with the INVALID option.

ELEMENT:

ACCEPT

FORMAT:

ACCEPT <identifier>

DESCRIPTION:

This statement reads up to 72 characters from the console. The usage of the item must be DISPLAY.

EXAMPLES:

ACCEPT IMAGE

ACCEPT NUM(9)

ELEMENT:

ADD

FORMAT:

```
ADD {identifier} [{identifier-1}] TO identifier-2  
    {literal}    {literal}
```

```
[ROUNDED] [SIZE ERROR <imperative-statement>].
```

DESCRIPTION:

This instruction adds either one or two numbers to a third with the result being placed in the last location.

EXAMPLES:

```
ADD 10 TO NUMB1
```

```
ADD X Y TO Z ROUNDED.
```

```
ADD 100 TO NUMBER SIZE ERROR GO ERROR-LOC
```


ELEMENT:

CALL

FORMAT:

CALL literal [USING name1 [name2] ... [name5]]

DESCRIPTION:

CALL is not implemented.

ELEMENT:

CLOSE

FORMAT:

CLOSE file-name

DESCRIPTION:

Files must be closed if they have been written. However, the normal requirement to close an input file prior to the end of processing does not exist.

EXAMPLES:

CLOSE FILE1

CLOSE RANDFILE

ELEMENT:

DELETE

FORMAT:

DELETE record-name [INVALID <imperative-statement>]

DESCRIPTION:

This statement requires the record name, not the file name as in the standard form of the statement. Since there is no deletion mark in CP/M, this would normally result in the record still being readable. It is, therefore, filled with zeroes to indicate that it has been removed.

EXAMPLES:

DELETE RECORD1

ELEMENT:

DISPLAY

FORMAT:

```
DISPLAY {identifier} [{identifier-1}]  
        {literal}    } {literal}    }
```

DESCRIPTION:

This displays the contents of an identifier or displays a literal on the console. Usage must be DISPLAY. The maximum length of the display is 72 positions.

EXAMPLES:

DISPLAY MESSAGE-1

DISPLAY MESSAGE-3 10

DISPLAY 'THIS MUST BE THE END'

ELEMENT:

DIVIDE

FORMAT:

DIVIDE {identifier} INTO identifier-1 [ROUNDED]
 {literal }

[SIZE ERROR <imperative-statement>]

DESCRIPTION:

The result of the division is stored in identifier-1;
any remainder is lost.

EXAMPLES:

DIVIDE NUMB INTO STORE

DIVIDE 25 INTO RESULT

ELEMENT:

ENTER

FORMAL:

ENTER language-name (routine-name)

DESCRIPTION:

This construct is not implemented.

ELEMENT:

EXIT

FORMAT:

EXIT [PROGRAM]

DESCRIPTION:

The EXIT command causes no action by the interpreter but allows for an empty paragraph for the construction of a common return point. The optional PROGRAM statement is not implemented as it is part of the interprogram communication module.

EXAMPLES:

RETURN.

EXIT.

ELEMENT:

GO

FORMAT:

1.

GO procedure-name

2.

GO procedure-1 [procedure-2] ... procedure-20
DEPENDING identifier

DESCRIPTION:

The go command causes an unconditional branch to the routine specified. The second form causes a forward branch depending on the value of the contents of the identifier. The identifier must be a numeric integer value. There can be no more than 20 procedure names.

EXAMPLES:

GO READ-CARD.

GO READ1 READ2 READ3 DEPENDING READ-INDEX.

ELEMENT:

IF

FORMAT:

```
IF <condition> {imperative  } ELSE imperative-2  
                {NEXT SENTENCE}
```

DESCRIPTION:

This is the standard COBOL IF statement. Note that there is no nesting of IF statements allowed since the IF statement is a conditional.

EXAMPLES:

```
IF A GREATER B ADD A TO C ELSE GO ERROR-ONE.
```

```
IF A NOT NUMERIC NEXT SENTENCE ELSE MOVE ZERO TO A.
```


ELEMENT:

MOVE

FORMAT:

```
MOVE {identifier-1} TO identifier-2  
    {literal      }
```

DESCRIPTION:

The standard list of allowable moves applies to this action. As a space saving feature of this implementation, all numeric moves go through the accumulators. This makes numeric moves slower than alpha-numeric moves, and where possible they should be avoided. Any move that involves picture clauses that are exactly the same can be accomplished as an alpha-numeric move if the elements are redefined as alpha-numeric; also all group moves are alpha-numeric.

EXAMPLES:

```
MOVE SPACE TO PRINT-LINE.
```

```
MOVE A(10) TO B(PTR).
```


ELEMENT:

MULTIPLY

FORMAT:

MULTIPLY {identifier} BY identifier-2 [ROUNDED]
 {literal }

 [SIZE ERROR <imperative-statement>]

DESCRIPTION:

The multiply routine requires enough space to calculate the result with the full number of decimal digits prior to moving the result into identifier-2. This means that a number with 5 places after the decimal multiplied by a number with 6 places after the decimal will generate a number with 11 decimal places which would overflow if there were more than 7 digits before the decimal place.

EXAMPLES:

MULTIPLY X BY Y.

MULTIPLY A BY B(7) SIZE ERROR GO OVERFLOW.

ELEMENT:

OPEN

FORMAT:

```
OPEN {INPUT file-name }  
      {OUTPUT file-name}  
      {I-O file-name   }
```

DESCRIPTION:

These three types of opens have the exact same effect on the diskette. However, they do allow for internal checking of the other file actions. For example, a write to a file set open as input will cause a fatal error.

EXAMPLES:

```
OPEN INPUT CARDS.
```

```
OPEN OUTPUT REPORT-FILE.
```


ELEMENT:

PERFORM

FORMAT:

1.

PERFORM procedure-name [THRU procedure-name-2]

2.

PERFORM procedure-name [THRU procedure-name-2]

{identifier} TIMES
{integer }

3.

PERFORM procedure-name [THRU procedure-name-2]

UNTIL <condition>

DESCRIPTION:

All three options are supported. Branching may be either forward or backward, and the procedures called may have perform statements in them as long as the end points do not coincide or overlap.

EXAMPLES:

PERFORM OPEN-ROUTINE.

PERFORM TOTALS THRU END-REPORT.

PERFORM SUM 10 TIMES.

PERFORM SKIP-LINE UNTIL PG-CNT GREATER 60.

ELEMENT:

READ

FORMAT:

1.

READ file-name INVALID <imperative-statement>

2.

READ file-name END <imperative-statement>

DESCRIPTION:

The invalid condition is only applicable to files in a random mode. All sequential files must have an END statement.

EXAMPLES:

READ CARDS END GO END-OF-FILE.

READ RANDOM-FILE INVALID MOVE SPACES TO REC-1.

ELEMENT:

REWRITE

FORMAT:

REWRITE file-name [INVALID <imperative>]

DESCRIPTION:

REWRITE is only valid for files that are open in the I-O mode. The INVALID clause is only valid for random files. This statement results in the current record being written back into the place that it was just read from. Note that this requires a file name not a record name.

EXAMPLES:

REWRITE CARDS.

REWRITE RAND-1 INVALID PERFORM ERROR-CHECK.

ELEMENT:

STOP

FORMAT:

```
STOP {RUN  }  
      {literal}
```

DESCRIPTION:

This statement ends the running of the interpreter. If a literal is specified, then the literal is displayed on the console prior to termination of the program.

EXAMPLES:

STOP RUN.

STOP 1.

STOP "INVALID FINISH".

ELEMENT:

SUBTRACT

FORMAT:

SUBTRACT {identifier-1} [identifier-2] FROM identifier-3
 {literal-1 } [literal-2]

[ROUNDED] [SIZE ERROR <imperative-statement>]

DESCRIPTION:

Identifier-3 is decremented by the value of identifier/literal one, and, if specified, identifier/literal two. The results are stored back in identifier-3. Rounding and size error options are available if desired.

EXAMPLES:

SUBTRACT 10 FROM SUB(12).

SUBTRACT A B FROM C ROUNDED.

ELEMENT:

WRITE

FORMAT:

1.

```
WRITE file-name [{BEFORE} ADVANCING {INTEGER}]  
                  {AFTER }                {PAGE  }
```

2.

```
WRITE file-name INVALID <imperative-statement>
```

DESCRIPTION:

There is no printer on the 8080 system here, so the ADVANCING option is not implemented. The INVALID option only applies to random files.

EXAMPLES:

```
WRITE OUT-FILE.
```

```
WRITE RAND-FILE INVALID PERFORM ERROR-RECOV.
```


ELEMENT:

<condition>

FORMAT:

RELATIONAL CONDITION:

```
{identifier-1} [NOT] {GREATER} {identifier-2}
{literal-1}      {LESS   } {literal-2   }
                  {EQUAL  }
```

CLASS CONDITION:

```
identifier [NOT] {NUMERIC   }
                  {ALPHABETIC}
```

DESCRIPTION:

It is not valid to compare two literals. The class condition NUMERIC will allow for a sign if the identifier is signed numeric.

EXAMPLES:

A NOT LESS 10.

LINE GREATER "C".

NUMB1 NOT NUMERIC

ELEMENT:

Subscripting

FORMAT:

data-name (subscript)

DESCRIPTION:

Any item defined with an OCCURS may be referenced by a subscript. The subscript may be a literal integer, or it may be a data item that has been specified as an integer. If the subscript is signed, the sign must be positive at the time of its use.

EXAMPLES:

A(10)

ITEM(SUB)

IV. COMPILER TOGGLES

There are four toggles in the compiler. They are entered on the first line of the program as a dollar sign followed by the given letter. In each case the toggle reverses the default value.

\$L -- list the input code on the screen as the program is compiled. Default is on. Error messages will be difficult to understand if this toggle is turned off, but if the interface device is a teletype, it may be desired in certain situations.

\$S -- sequence numbers are in the first six positions of each record. Default is off.

\$P -- list productions as they occur. Default is off.

\$T -- list tokens from the scanner. Default is off.

V. RUN TIME CONVENTIONS

This section explains how to run the compiler on the current system. The compiler expects to see a file with a type of CBL as the input file. In general, the input is free form. If the input includes line numbers then the compiler must be notified by setting the appropriate toggle. The compiler is started by typing COBOL <file-name>. Where the file name is the system name of the input file. There is no interaction required to start the second part of the compiler. The output file will have the same file name as the input file, and will be given a file type of CIM. Any previous copies of the file will be erased.

The interpreter is started by typing CBLINT <file-name>. The first program is a loader, and it will display "LOAD FINISHED" to indicate successful completion. The run-time package will be brought in by the build program, and execution should continue without interruption.

VI. FILE INTERACTIONS WITH CP/M

The file structure that is expected by the program imposes some restrictions on the system. References 2 and 3 contain detailed information on the facilities of CP/M, and should be consulted for details. The information that has been included in this section is intended to explain where limitations exist and how the program interacts with the system.

All files in CP/M are on a random access device, and there is no way for the system to distinguish sequential files from files created in a random mode. This means that the various types of reads and writes are all valid to any file that has fixed length records. The restrictions of the ASSIGN statement do prevent a file from being open for both random and sequential actions during one program.

Each logical record is terminated by a carriage return and a line feed. In the case of variable length records, this is the only end mark that exists. This convention was adopted to allow the various programs which are used in CP/M to work with the files. Files created by the editor, for example, will generally be variable length files. This convention does remove the capability of reading variable length files in a random mode.

All of the physical records are assumed to be 128 bytes in length, and the program supplies buffer space for these records in addition to the logical records. Logical records may be of any desired length.

ERROR MESSAGES

COMPILER FATAL MESSAGES

BR Bad read -- disk error, no corrective action can be taken in the program.

CL Close error -- unable to close the output file.

MA Make error -- could not create the output file.

MO Memory overflow -- the code and constants generated will not fit in the allotted memory space.

OP Open error -- can not open the input file, or no such file present.

ST Symbol table overflow -- symbol table is too large for the allocated space.

WR Write error -- disk error, could not write a code record to the disk.

COMPILER WARNINGS

EL Extra levels -- only 10 levels are allowed.

- FT File type -- the data element used in a read or write statement is not a file name.
- IA Invalid access -- the specified options are not an allowable combination.
- ID Identifier stack overflow -- more than 20 items in a GO TO -- DEPENDING statement.
- IS Invalid subscript -- an item was subscripted but it was not defined by an OCCURS.
- IT Invalid type -- the field types do not match for this statement.
- LE Literal error -- a literal value was assigned to an item that is part of a group item previously assigned a value.
- NF No file assigned -- there was no SELECT clause for this file.
- NI Not implimented -- a production was used that is not implimented.
- NN Non-numeric -- an invalid character was found in a numeric string.

- NP No production -- no production exists for the current parser configuration; error recovery will automatically occur.
- NV Numeric value -- a numeric value was assigned to a non-numeric item.
- PC Picture clause -- an invalid character or set of characters exists in the picture clause.
- PF Paragraph first -- a section header was produced after a paragraph header, which is not in a section.
- R1 Redefine nesting -- a redefinition was made for an item which is part of a redefined item.
- R2 Redefine length -- the length of the redefinition item was greater than the item that it redefined.
- SE Scanner error -- the scanner was unable to read an identifier due to an invalid character.
- SG Sign error -- either a sign was expected and not found, or a sign was present when not valid.
- SL Significance loss -- the number assigned as a value is larger than the field defined.

TE Type error -- the type of a subscript index is not integer numeric.

VE Value error -- a value statement was assigned to an item in the file section.

INTERPRETER FATAL ERRORS

CL Close error -- the system was unable to close an output file.

ME Make error -- the system was unable to make an input file on the disk.

NF No file -- an input file could not be opened.

WI Write to input -- a write was attempted to an input file.

INTERPRETER WARNING MESSAGES

EM End mark -- a record that was read did not have a carriage return or a line feed in the expected location.

GD Go to depending -- the value of the depending indicator was greater than the number of available branch

addresses.

IC Invalid character -- an invalid character was loaded into an output field during an edited move. For example, a numeric character into an alphabetic-only field.

SI Sign Invalid -- the sign is not a "+" or a "-".

LIST OF REFERENCES

1. Craig, A. S. MICRO-COBOL an implementation of Navy Standard HYPO-COBOL for a microprocessor-based computer system, Masters Thesis, Naval Postgraduate School, March 1977.
2. Digital Research, An Introduction to CP/M Features and Facilities, 1976
3. Digital Research, CP/M Interface Guide, 1976.
4. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.
5. Intel Corporation, 8080 Simulator Software Package, 1974.
6. Software Development Division, ADPE Selection Office, Department of the Navy, HYPO-COBOL, April 1975.
7. Strutynski, Kathryn B. Information on the CP/M Interface Simulator, internally distributed technical note.

[illegible]


```

00109 1 MCN1: PROCEDURE (F,A);
00110 1 DECLARE F BYTE, A ADDRESS;
00111 2 CC TO BCCS;
00112 2 END MCN1;
00113 1
00114 1 MCN2: PROCEDURE (F,A) BYTE;
00115 1 DECLARE F BYTE, A ADDRESS;
00116 2 CC TO BCCS;
00117 2 END MCN2;
00118 1
00119 1 PRINTCHAR: PROCEDURE (CHAR);
00120 1 DECLARE CHAR BYTE;
00121 2 CALL MCN1 (2,CHAR);
00122 2 END PRINTCHAR;
00123 1
00124 1 CRLF: PROCEDURE;
00125 1 CALL PRINTCHAR(CR);
00126 2 CALL PRINTCHAR(LF);
00127 2 END CRLF;
00128 1
00129 1 PRINT: PROCEDURE (A);
00130 1 DECLARE A ADDRESS;
00131 2 CALL MCN1 (9,A);
00132 2 END PRINT;
00133 1
00134 1 PRINT$ERROR: PROCEDURE (CODE);
00135 1 DECLARE CODE ADDRESS;
00136 2 CALL CRLF;
00137 2 CALL PRINTCHAR(HIGH(CODE));
00138 2 CALL PRINTCHAR(LOW(CODE));
00139 2 END PRINT$ERROR;
00140 1
00141 1 FATAL$ERROR: PROCEDURE(REASON);
00142 1 DECLARE REASON ADDRESS;
00143 2 CALL PRINT$ERROR(REASON);
00144 2 CALL TIME(10);
00145 2 GO TO BCOT;
00146 2 END FATAL$ERROR;
00147 1
00148 1 OPEN: PROCEDURE;
00149 2 IF MON2 (15,IN$ADDR)=255 THEN CALL FATAL$ERROR('OP');
00150 2 END OPEN;
00151 1
00152 1 MCRE$INPUT: PROCEDURE BYTE;
00153 2 /* READS THE INPUT FILE AND RETURNS TRUE IF A RECCRD
00154 2 WAS REAC. FALSE IMPLIES END OF FILE */
00155 2 DECLARE DCNT BYTE;
00156 2 IF (DCNT:=MCN2(20,.INPUT$FCB))>1 THEN CALL FATAL$ERROR('BR');
00157 2 RETURN NOT(DCNT);
00158 2 END MCRE$INPUT;
00159 1
00160 1 MAKE: PROCEDURE;
00161 2 /* DELETES ANY EXISTING COPY OF THE OUTPUT FILE
00162 2 AND CREATES A NEW COPY*/
00163 2 CALL MCN1(19,.OUTPUT$FCB);
00164 2 IF MON2(22,.OUTPUT$FCB)=255 THEN CALL FATAL$ERROR('MA');
00165 2 END MAKE;
00166 1
00167 1 WRITE$OUTPUT: PROCEDURE;
00168 2 /* WRITES OUT A BUFFER */
00169 2 CALL MCN1(26,.OUTPUT$BUFF); /* SET DMA */
00170 2 IF MON2(21,.OUTPUT$FCB)<>0 THEN CALL FATAL$ERROR('WR');
00171 2 CALL MCN1(26,BCH); /* RESET DMA */
00172 2 END WRITE$OUTPUT;
00173 1
00174 1 MCVE: PROCEDURE (SOURCE, DESTINATION, COUNT);
00175 2 /* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
00176 2 DECLARE (SOURCE,DESTINATION) ADDRESS;
00177 2 ($$BYTE BASED SOURCE, $$BYTE BASED DESTINATION, COUNT) BYTE;
00178 2 DO WHILE (COUNT:=COUNT - 1)<> 255;
00179 3 D$BYTE=$$BYTE;
00180 3 SOURCE=SOURCE + 1;
00181 3 DESTINATION = DESTINATION + 1;
00182 3 END;
00183 2 END MCVE;
00184 1
00185 1 FILL: PROCEDURE(ADDR,CHAR,COUNT);
00186 2 /* MOVES CHAR INTO ADDR FOR COUNT BYTES */
00187 2 DECLARE ADDR ADDRESS;
00188 2 (CHAR,COUNT,DEST BASED ADDR) BYTE;
00189 2 DO WHILE (COUNT:=COUNT -1)<>255;
00190 3 DEST=CHAR;
00191 3 ADDR=ADDR + 1;
00192 3 END;
00193 2 END FILL;
00194 1
00195 1 /* * * * * * SCANNER LITS * * * * */
00196 1 DECLARE
00197 1 LITERAL LIT '15';
00198 1 INPUT$STR LIT '32';
00199 1 PERIOD LIT '1';
00200 1 INVALID LIT '0';
00201 1
00202 1 /* * * * * SCANNER TABLES * * * * */
00203 1 DECLARE TOKEN$TABLE DATA
00204 1 /* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
00205 1 FOR EACH LENGTH OF WORD */
00206 1 (0,0,1,4,5,15,22,32,38,44,47,49,51,55,56,57);
00207 1
00208 1 TABLE DATA('FC','OF','TO','PIC','COMP','DATA','FILE'
00209 1 , 'LEFT','MCDE','SAME','SIGN','SYNC','ZERO','BLOCK','LABEL'
00210 1 , 'CUTER','RIGHT','SPACE','USAGE','VALUE','ACCESS','ASSIGN'
00211 1 , 'AUTHOR','FILL$P','OCCURS','RANDOM','RECCRD','SELECT'
00212 1 , 'DISPLAY','LEADING','LINKAGE','OMITTED','RECORDS'
00213 1 , 'SECTION','DIVISION','RELATIVE','SECURITY','SEPARATE','STANDARD'
00214 1 , 'TRAILING','DEBUGGING','PROCEDURE','REDEFINES'
00215 1 , 'PROGRAM-10','SEQUENTIAL','ENVIRONMENT','I-O-CONTROL'
00216 1 , 'DATE-WRITTEN','FILE-CONTROL','INPUT-OUTPUT','ORGANIZATION'
00217 1
00218 1

```



```

00219 1      , 'CONFIGURATION', 'IDENTIFICATION', 'OBJECT-COMPUTER'
00220 1      , 'SOURCE-COMPUTER', 'WORKING-STORAGE' ),
00221 1
00222 1  OFFSET (16) ADDRESS
00223 1      /* NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
00224 1      INITIAL (0,0,0,6,9,45,80,128,170,218,245,265,
00225 1      287,335,346,362),
00226 1
00227 1  WORD$COUNT DATA
00228 1      /* NUMBER OF WORDS OF EACH SIZE */
00229 1      (0,0,3,1,9,7,8,6,6,3,2,2,4,1,1,3),
00230 1
00231 1
00232 1      MAX$LEN      LIT      '16',
00233 1      ADD$END      DATA    ('PROCEDURE '),
00234 1      LOCKED       BYTE     INITIAL (0),
00235 1      HOLD         BYTE
00236 1      BUFFER$END   ADDRESS   INITIAL (100H),
00237 1      NEXT         BASED     POINTER BYTE,
00238 1      INBUFF       LIT      '80H',
00239 1      CHAR         BYTE,
00240 1      ACCUM$LENG   LIT      '50',
00241 1      ACCUM        BYTE,
00242 1      R$ACCUM      (ACCUM$LENG) BYTE,
00243 1      DISPLAY      BYTE     INITIAL (0),
00244 1      DISPLAY$REST (7)      BYTE,
00245 1      TCKEN        BYTE;    /*RETURNED FROM SCANNER */
00246 1
00247 1      /* * * * * PROCEDURES USED BY THE SCANNER * * * */
00248 1
00249 1  NEXT$CHAR: PROCEDURE BYTE;
00250 1      IF LOCKED THEN
00251 1          CC;
00252 1          LOCKED=FALSE;
00253 1          RETURN (CHAR:=HOLD);
00254 1      END;
00255 1      IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
00256 1          CC;
00257 1          IF NOT MORE$INPUT THEN
00258 1              DC;
00259 1              BUFFER$END=.MEMORY;
00260 1              POINTER=.ADD$END;
00261 1              END;
00262 1              ELSE POINTER=INBUFF;
00263 1              END;
00264 1              RETURN (CHAR:=NEXT);
00265 1      END NEXT$CHAR;
00266 1
00267 1  GET$CHAR: PROCEDURE;
00268 1      /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
00269 1      THE DIRECT RETURN OF THE CHARACTER*/
00270 1      CHAR=NEXT$CHAR;
00271 1      END GET$CHAR;
00272 1
00273 1  DISPLAY$LINE: PROCEDURE;
00274 1      IF NOT LIST$INPUT THEN RETURN;
00275 1      DISPLAY(DISPLAY + 1) = ' ';
00276 1      CALL PRINT(.DISPLAY$REST);
00277 1      DISPLAY=0;
00278 1      END DISPLAY$LINE;
00279 1
00280 1  LOAD$DISPLAY: PROCEDURE;
00281 1      IF DISPLAY < 72 THEN
00282 1          DISPLAY(DISPLAY:=DISPLAY + 1) = CHAR;
00283 1          CALL GET$CHAR;
00284 1      END LOAD$DISPLAY;
00285 1
00286 1  PLT: PROCEDURE;
00287 1      IF ACCUM < ACCUM$LENG THEN
00288 1          ACCUM(ACCUM:=ACCUM+1)=CHAR;
00289 1          CALL LOAD$DISPLAY;
00290 1      END PLT;
00291 1
00292 1  EAT$LINE: PROCEDURE;
00293 1      CC WHILE CHAR<>CR;
00294 1      CALL LOAD$DISPLAY;
00295 1      END;
00296 1      END EAT$LINE;
00297 1
00298 1  GET$NC$BLANK: PROCEDURE;
00299 1      DECLARE (N,I) BYTE;
00300 1      CC FOREVER;
00301 1          IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
00302 1          ELSE
00303 1              IF CHAR=CR THEN
00304 1                  CC;
00305 1                  CALL DISPLAY$LINE;
00306 1                  IF SEQ$NUM THEN N=3; ELSE N=2;
00307 1                  CC I = 1 TO N;
00308 1                      CALL LOAD$DISPLAY;
00309 1                  END;
00310 1                  IF CHAR = '*' THEN CALL EAT$LINE;
00311 1                  ELSE
00312 1                      IF CHAR = ':' THEN
00313 1                          DC;
00314 1                          IF NOT DEBUGGING THEN CALL EAT$LINE;
00315 1                          ELSE CALL LOAD$DISPLAY;
00316 1                      END;
00317 1                  END;
00318 1                  RETURN;
00319 1              ELSE
00320 1                  RETURN;
00321 1              END;
00322 1      END; /* END OF DC FOREVER */
00323 1  END GET$NC$BLANK;
00324 1
00325 1  SPACE: PROCEDURE BYTE;
00326 1      RETURN (CHAR=' ') OR (CHAR=CR);
00327 1  END SPACE;
00328 1

```



```

00329 1 DELIMITER: PROCEDURE BYTE;
00330 2 /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR */
00331 2 IF CHAR <> ' ' THEN RETURN FALSE;
00332 2 PCU=NEXT$CHAR;
00333 2 LOCKED=TRUE;
00334 2 IF SPACE THEN
00335 2   CC;
00336 2   CHAR = ' ';
00337 2   RETURN TRUE;
00338 2 END;
00339 2 CHAR='.';
00340 2 RETURN FALSE;
00341 2 END DELIMITER;
00342 1
00343 1 END$CF$TOKEN: PROCEDURE BYTE;
00344 1 RETURN SPACE OR DELIMITER;
00345 2 END END$CF$TOKEN;
00346 1
00347 1 GET$LITERAL: PROCEDURE BYTE;
00348 1 CALL LCAD$DISPLAY;
00349 1 CC FOREVER;
00350 1 IF CHAR= QUOTE THEN
00351 1   CC;
00352 1   CALL LCAD$DISPLAY;
00353 1   RETURN LITERAL;
00354 1   END;
00355 1   CALL PUT;
00356 1 END;
00357 1 END GET$LITERAL;
00358 1
00359 1
00360 1 LOCK$UP: PROCEDURE BYTE;
00361 1 DECLARE PCINT ADDRESS;
00362 1 (HERE BASED PCINT,I) BYTE;
00363 1
00364 1 MATCH: PROCEDURE BYTE;
00365 1 DECLARE J BYTE;
00366 1 DO J=1 TO ACCUM;
00367 1   IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
00368 1   END;
00369 1   RETURN TRUE;
00370 1 END MATCH;
00371 1
00372 1 POINT=OFFSET(ACCUM)+.TABLE;
00373 1 DO I=1 TO WORD$COUNT(ACCUM);
00374 1   IF MATCH THEN RETURN I;
00375 1   PCINT = PCINT + ACCUM;
00376 1   END;
00377 1   RETURN FALSE;
00378 1 END LOCK$UP;
00379 1
00380 1
00381 1 RESERVED$WORD: PROCEDURE BYTE;
00382 1 /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
00383 1 THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
00384 1 DECLARE VALUE BYTE;
00385 1 DECLARE NUMB BYTE;
00386 1 IF ACCUM > MAX$LEN THEN RETURN 0;
00387 1 IF (NUMB:=TOKEN$TABLE(ACCUM))=0 THEN RETURN 0;
00388 1 IF (VALUE:=LOCK$UP)=0 THEN RETURN 0;
00389 1 RETURN (NUMB + VALUE);
00390 1 END RESERVED$WORD;
00391 1
00392 1
00393 1 GET$TOKEN: PROCEDURE BYTE;
00394 1 ACCUM=0;
00395 1 CALL GET$AC$BLANK;
00396 1 IF CHAR=QUOTE THEN RETURN GET$LITERAL;
00397 1 IF DELIMITER THEN
00398 1   CC;
00399 1   CALL PUT;
00400 1   RETURN PERIOD;
00401 1   END;
00402 1   CC FOREVER;
00403 1   CALL PUT;
00404 1   IF END$CF$TOKEN THEN RETURN INPUT$STR;
00405 1   END; /* OF CC FOREVER */
00406 1 END GET$TOKEN;
00407 1
00408 1
00409 1 SCANNER: PROCEDURE;
00410 1 DECLARE CHECK BYTE;
00411 1 CC FOREVER;
00412 1 IF (TOKEN:=GET$TOKEN) = INPUT$STR THEN
00413 1   IF (CHECK:=RESERVED$WORD) <> 0 THEN TOKEN=CHECK;
00414 1   CALL TOKEN <> 0 THEN RETURN;
00415 1   CALL PRINT$ERROR('S');
00416 1   DO WHILE NOT END$OF$TOKEN;
00417 1     CALL GET$CHAR;
00418 1   END;
00419 1 END SCANNER;
00420 1
00421 1
00422 1 PRINT$ACCU: PROCEDURE;
00423 1 ACCUM(ACCUM+1)=' ';
00424 1 CALL PRINT(,R$ACCUM);
00425 1 END PRINT$ACCU;
00426 1
00427 1 PRINT$NUMBER: PROCEDURE(NUMB);
00428 1 DECLARE (NUMB,I,CNT,K) BYTE, J DATA(100,10);
00429 1 DO I=0 TO 1;
00430 1   CNT=0;
00431 1   DO WHILE NUMB >= (K:=J(I));
00432 1     NUMB=NUMB - K;
00433 1     CNT=CNT + 1;
00434 1   END;
00435 1   CALL PRINT$CHAR('0' + CNT);
00436 1   END;
00437 1   CALL PRINT$CHAR('0' + NUMB);
00438 1 END PRINT$NUMBER;

```



```

00439 1
00440 1
00441 1
00442 1
00443 2 INIT$SCANNER: PROCEDURE;
00444 2 /* INITIALIZE FOR INPUT - OUTPUT OPERATIONS */
00445 2 CALL MOVE ('CBL', IN$ADDR + 9, 3);
00446 2 CALL FILL (IN$ADDR + 12, 0, 5);
00447 2 CALL OPEN;
00448 2 CALL MOVE (IN$ADDR, OUTPUT$FCB, 9);
00449 2 OUTPUT$ENO = (OUTPUT$PTR - OUTPUT$BUFF - 1) + 128;
00450 2 CALL MAKE;
00451 2 CALL GET$CHAR; /* PRIME THE SCANNER */
00452 2 DO WHILE CHAR = '$';
00453 2 IF NEXTCHAR = 'L' THEN LIST$INPUT = NOT LIST$INPUT;
00454 2 ELSE IF CHAR = 'S' THEN SEQ$NUM = NOT SEQ$NUM;
00455 2 ELSE IF CHAR = 'P' THEN PRINT$PROD = NOT PRINT$PROD;
00456 2 ELSE IF CHAR = 'T' THEN PRINT$TOKEN = NOT PRINT$TOKEN;
00457 2 CALL GET$CHAR;
00458 2 CALL GET$NO$BLANK;
00459 2 END;
00460 1 END INIT$SCANNER;
00461 1 /* * * * * END OF SCANNER PROCEDURES * * * */
00462 1
00463 1 /* * * * * SYMBOL TABLE DECLARATIONS * * * */
00464 1
00465 1 DECLARE
00466 1
00467 1 CUR$SYM ADDRESS, /*SYMBOL BEING ACCESSED*/
00468 1 SYMBOL BASED CUR$SYM BYTE,
00469 1 SYMBOL$ADDR BASED CUR$SYM ADDRESS,
00470 1 NEXT$SYM$ENTRY BASED NEXT$SYM ADDRESS,
00471 1 HASH$PTR ADDRESS,
00472 1 DISPLACEMENT LIT '12',
00473 1 HASH$MASK LIT '3FH',
00474 1 $TYPE LIT '2',
00475 1 $C$CURS LIT '11',
00476 1 $ADDR2 LIT '4',
00477 1 $LENGTH LIT '3',
00478 1 $LENGTH LIT '3',
00479 1 LEVEL LIT '10',
00480 1 LOCATION LIT '2',
00481 1 REL$IC LIT '5',
00482 1 START$NAME LIT '11', /*1 LESS*/
00483 1 MAX$IC$LEN LIT '12';
00484 1
00485 1 /* * * * * TYPE LITERALS * * * * */
00486 1
00487 1 DECLARE
00488 1 SEQUENTIAL LIT '1',
00489 1 RANDCM LIT '2',
00490 1 SEQ$RELATIVE LIT '3',
00491 1 VARIABLE$LENG LIT '4',
00492 1 GR$UP LIT '6',
00493 1 CCMP LIT '21';
00494 1
00495 1 /* * * * * SYMBOL TABLE ROUTINES * * * */
00496 1
00497 1 INIT$SYMBOL: PROCEDURE;
00498 1 CALL FILL (FREE$STORAGE, 0, 130);
00499 1 /* INITIALIZE HASH TABLE AND FIRST COLLISION FIELD */
00500 1 NEXT$SYM = FREE$STORAGE + 128;
00501 1 NEXT$SYM$ENTRY = 0;
00502 1 END INIT$SYMBOL;
00503 1
00504 1 GET$P$LENGTH: PROCEDURE BYTE;
00505 1 RETURN SYMBOL$ADDR(P$LENGTH);
00506 1 END GET$P$LENGTH;
00507 1
00508 1 SET$ADDRESS: PROCEDURE (ADDR);
00509 1 DECLARE ADDR ADDRESS;
00510 1 SYMBOL$ADDR(LOCATION) = ADDR;
00511 1 END SET$ADDRESS;
00512 1
00513 1 GET$ADDRESS: PROCEDURE ADDRESS;
00514 1 RETURN SYMBOL$ADDR(LOCATION);
00515 1 END GET$ADDRESS;
00516 1
00517 1 GET$TYPE: PROCEDURE BYTE;
00518 1 RETURN SYMBOL($TYPE);
00519 1 END GET$TYPE;
00520 1
00521 1 SET$TYPE: PROCEDURE (TYPE);
00522 1 DECLARE TYPE BYTE;
00523 1 SYMBOL($TYPE) = TYPE;
00524 1 END SET$TYPE;
00525 1
00526 1 OR$TYPE: PROCEDURE (TYPE);
00527 1 DECLARE TYPE BYTE;
00528 1 SYMBOL($TYPE) = TYPE OR GET$TYPE;
00529 1 END OR$TYPE;
00530 1
00531 1 GET$LEVEL: PROCEDURE BYTE;
00532 1 RETURN SHR(SYMBOL(LEVEL), 4);
00533 1 END GET$LEVEL;
00534 1
00535 1 SET$LEVEL: PROCEDURE (LVL);
00536 1 DECLARE LVL BYTE;
00537 1 SYMBOL(LEVEL) = SHL(LVL, 4) OR SYMBOL(LEVEL);
00538 1 END SET$LEVEL;
00539 1
00540 1 GET$DECIMAL: PROCEDURE BYTE;
00541 1 RETURN SYMBOL(LEVEL) AND OFH;
00542 1 END GET$DECIMAL;
00543 1
00544 1 SET$DECIMAL: PROCEDURE (DEC);
00545 1 DECLARE DEC BYTE;
00546 1 SYMBOL(LEVEL) = DEC OR SYMBOL(LEVEL);
00547 1 END SET$DECIMAL;
00548 1

```



```

00545 1 SET$S$LENGTH: PROCEDURE (HOW$LONG);
00550 1 DECLARE HOW$LONG ADDRESS;
00551 2 SYMBOL$ADDR(S$LENGTH) = HOW$LONG;
00552 2 END SET$S$LENGTH;
00553 2
00554 1 GET$S$LENGTH: PROCEDURE ADDRESS;
00555 1 RETURN SYMBOL$ADDR(S$LENGTH);
00556 2 END GET$S$LENGTH;
00557 2
00558 1
00559 1 SET$ADDR2: PROCEDURE (ADDR);
00560 1 DECLARE ADDR ADDRESS;
00561 2 SYMBOL$ADDR(ADDR2)=ADDR;
00562 2 END SET$ADDR2;
00563 2
00564 1 GET$ADDR2: PROCEDURE ADDRESS;
00565 1 RETURN SYMBOL$ADDR(ADDR2);
00566 2 END GET$ADDR2;
00567 2
00568 1
00569 1 SET$CCCURS: PROCEDURE (CCCUR);
00570 1 DECLARE CCUR BYTE;
00571 2 SYMBOL$(CCCURS)=CCCUR;
00572 2 END SET$CCCURS;
00573 1
00574 1 GET$CCCURS: PROCEDURE BYTE;
00575 1 RETURN SYMBOL$(CCCURS);
00576 2 END GET$CCCURS;
00577 2
00578 1 /* * * * * PARSE DECLARATIONS * * * */
00579 1 DECLARE
00580 1 INT LIT '63', /* CODE FOR INITIALIZE */
00581 1 SCD LIT '66', /* CODE FOR SET CODE START */
00582 1 PSTACKSIZE LIT '30', /* SIZE OF PARSE STACKS */
00583 1 STATESTACK (PSTACKSIZE) BYTE, /* SAVED STATES */
00584 1 VALLE (PSTACKSIZE) ADDRESS, /* TEMP VALUES */
00585 1 VARC (51) BYTE, /* TEMP CHAR STORE */
00586 1 IC$STACK (10) ADDRESS, INITIAL (0),
00587 1 IC$STACK$PTR BYTE, INITIAL (0),
00588 1 HOLD$LIT BYTE,
00589 1 REST$HOLD$LIT (ACCU$LENG) BYTE,
00590 1 HCLC$SYM ADDRESS,
00591 1 PEN$ING$LIT$ID BYTE, INITIAL (FALSE),
00592 1 PEN$ING$LIT$ID ADDRESS,
00593 1 REDEF$ONE BYTE, INITIAL (FALSE),
00594 1 REDEF$ONE ADDRESS,
00595 1 REDEF$TWO ADDRESS,
00596 1 TEMP$HOLD ADDRESS,
00597 1 TEMP$TWO ADDRESS,
00598 1 CCM$FILLING BYTE, INITIAL (TRUE),
00599 1 SP BYTE, INITIAL (255),
00600 1 MP BYTE,
00601 1 MPP1 BYTE,
00602 1 NCL$CK BYTE, INITIAL (TRUE),
00603 1 (I,J,K) BYTE, /* INDICES FOR THE PARSER */
00604 1 STATE BYTE, INITIAL (STARTS);
00605 1
00606 1 /* * * * * PARSE ROUTINES * * * */
00607 1
00608 1 BYTE$OUT: PROCEDURE (CNE$BYTE);
00609 1 /* THIS PROCEDURE WRITES ONE BYTE OF OUTPUT ONTO THE DISK
00610 1 IF REQUIRED THE OUTPUT BUFFER IS DUMPED TO THE DISK */
00611 1 DECLARE CNE$BYTE BYTE;
00612 1 IF (OUTPUT$PTR=OUTPUT$PTR + 1) > OUTPUT$END THEN
00613 1 CC;
00614 1 CALL WRITES$OUTPUT;
00615 1 OUTPUT$PTR=OUTPUT$BUFF;
00616 1 END;
00617 1 OUTPUT$CHAR=CNE$BYTE;
00618 1 END BYTE$OUT;
00619 1
00620 1 STRING$OUT: PROCEDURE (ADDR,COUNT);
00621 1 DECLARE (ADDR,I,COUNT) ADDRESS, (CHAR BASEC ADDR) BYTE;
00622 1 CC I=1 TO COUNT;
00623 1 CALL BYTE$OUT(CHAR);
00624 1 ADDR=ADDR+1;
00625 1 END;
00626 1 END STRING$OUT;
00627 1
00628 1 ADDR$OUT: PROCEDURE (ADDR);
00629 1 DECLARE ADDR ADDRESS;
00630 1 CALL BYTE$OUT(LOW(ADDR));
00631 1 CALL BYTE$OUT(HIGH(ADDR));
00632 1 END ADDR$OUT;
00633 1
00634 1 FILL$STRING: PROCEDURE (COUNT,CHAR);
00635 1 DECLARE (I,COUNT) ADDRESS, CHAR BYTE;
00636 1 CC I=1 TO COUNT;
00637 1 CALL BYTE$OUT(CHAR);
00638 1 END;
00639 1 END FILL$STRING;
00640 1
00641 1 START$INITIALIZE: PROCEDURE (ADDR,CNT);
00642 1 DECLARE (ADDR,CNT) ADDRESS;
00643 1 CALL BYTE$OUT(INT);
00644 1 CALL ADDR$OUT(ADDR);
00645 1 CALL ADDR$OUT(CNT);
00646 1 END START$INITIALIZE;
00647 1
00648 1 BUILD$SYMBOL: PROCEDURE (LEN);
00649 1 DECLARE LEN BYTE, TEMP ADDRESS;
00650 1 TEMP=NEXT$SYM;
00651 1 IF (NEXT$SYM=SYMBOL(LEN=LEN+DISPLACEMENT))
00652 1 > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
00653 1 CALL FILL (TEMP,C,LEN);
00654 1 END BUILD$SYMBOL;

```



```

00655 1
00656 1
00657 1 MATCH: PROCEDURE ADDRESS;
00658 1 /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
00659 1 TABLE. IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS.
00660 1 OTHERWISE A NEW ENTRY IS MADE AND THE PRINT NAME
00661 1 IS ENTERED. ALL NAMES ARE TRUNCATED TO MAX$IOSLEN*/
00662 1 DECLARE (POINT, COLLISION BASED POINT) ADDRESS;
00663 1 (HCLO, I) BYTE;
00664 1 IF VARC>MAX$IOSLEN
00665 1 THEN VARC = MAX$IOSLEN;
00666 1 /* TRUNCATE IF REQUIRED */
00667 1 HCLO = C;
00668 1 CC I=1 TO VARC; /* CALCULATE HASH CODE */
00669 1 HCLO=HCLO + VARC(I);
00670 1 END;
00671 1 POINT=FREE$STORAGE + SHL((HOLD AND HASH$MASK),1);
00672 1 CC FOREVER;
00673 1 IF COLLISION=0 THEN
00674 1 OC;
00675 1 CUR$SYM, COLLISION=NEXT$SYM;
00676 1 CALL BUILD$SYMBOL(VARC);
00677 1 /* LOAD PRINT NAME */
00678 1 SYMBOL(P$LENGTH)=VARC;
00679 1 CO I = 1 TO VARC;
00680 1 SYMBOL(START$NAME + I)=VARC(I);
00681 1 END;
00682 1 RETURN CUR$SYM;
00683 1 ELSE
00684 1 DC;
00685 1 CUR$SYM=COLLISION;
00686 1 IF (HOLD:=GET$P$LENGTH)=VARC THEN
00687 1 CC;
00688 1 I=1;
00689 1 DO WHILE SYMBOL(START$NAME + I)=VARC(I);
00690 1 IF (I:=I+1)>HOLD THEN RETURN (CUR$SYM:=COLLISION);
00691 1 END;
00692 1 END;
00693 1 END;
00694 1 POINT=COLLISION;
00695 1 END;
00696 1 END MATCH;
00697 1
00698 1 ALLCATE: PROCEDURE (BYTES$REQ) ADDRESS;
00699 1 /* THIS ROUTINE CONTROLS THE ALLOCATION OF SPACE
00700 1 IN THE MEMORY OF THE INTERPRETER. */
00701 1
00702 1 DECLARE (HOLD, BYTES$REQ) ADDRESS;
00703 1 HCLO=NEXT$AVAILABLE;
00704 1 IF (NEXT$AVAILABLE:=NEXT$AVAILABLE + BYTES$REQ)>MAX$INT$MEM
00705 1 THEN CALL FATAL$ERROR('MO');
00706 1 RETURN HCLO;
00707 1 END ALLCATE;
00708 1
00709 1 SET$REDEF: PROCEDURE (CLO, NEW);
00710 1 DECLARE (CLO, NEW) ADDRESS;
00711 1 IF (REDEF:=NOT REDEF) THEN
00712 1 DC;
00713 1 REDEF$CNE=OLD;
00714 1 REDEF$TAC=NEW;
00715 1 END;
00716 1 ELSE CALL PRINT$ERRCR('R1');
00717 1 END SET$REDEF;
00718 1
00719 1 SET$CUR$SYM: PROCEDURE;
00720 1 CUR$SYM=ID$STACK(ID$STACK$PTR);
00721 1 END SET$CUR$SYM;
00722 1
00723 1 STACK$LEVEL: PROCEDURE BYTE;
00724 1 CALL SET$CUR$SYM;
00725 1 RETURN GET$LEVEL;
00726 1 END STACK$LEVEL;
00727 1
00728 1 LCAD$LEVEL: PROCEDURE;
00729 1 DECLARE HCLO ADDRESS;
00730 1
00731 1 LCAD$PEDEF$ACOR: PROCEDURE;
00732 1 CUR$SYM=REDEF$CNE;
00733 1 HCLO=GET$ADDRESS;
00734 1 END LCAD$REDEF$ACOR;
00735 1
00736 1 IF ID$STACK<>0 THEN
00737 1 DC;
00738 1 IF VALUE(SP-2)=0 THEN
00739 1 DC;
00740 1 CALL SET$CUR$SYM;
00741 1 HCLO=GET$S$LENGTH + GET$ADDRESS;
00742 1 END;
00743 1 ELSE CALL LCAD$REDEF$ACOR;
00744 1 IF (ID$STACK$PTR:=ID$STACK$PTR+1)>9 THEN
00745 1 DC;
00746 1 CALL PRINT$ERRCR('EL');
00747 1 ID$STACK$PTR=9;
00748 1 END;
00749 1 END;
00750 1 ELSE HCLO=NEXT$AVAILABLE;
00751 1 ID$STACK(ID$STACK$PTR)=VALUE(MPP1);
00752 1 CALL SET$CUR$SYM;
00753 1 CALL SET$ACCESS(HCLO);
00754 1 END LCAD$LEVEL;
00755 1

```



```

00756 1 REDEF10R$VALUE: PROCEDURE;
00757 DECLARE FCLC ADDRESS;
00758 (DEC, J, SIGN) BYTE;
00759 IF REDEF THEN
00760 CC;
00761 IF REDEF$TWC=CUR$SYM THEN
00762 CC;
00763 HOLD=GET$S$LENGTH;
00764 CUR$SYM=REDEF$ONE;
00765 IF HOLD>GET$S$LENGTH THEN
00766 CC;
00767 CALL PRINT$ERRJR('R2');
00768 HOLD=GET$S$LENGTH;
00769 CUR$SYM=REDEF$ONE;
00770 CALL SET$S$LENGTH(HOLD);
00771 END;
00772 REDEF=FALSE;
00773 END;
00774 END;
00775 IF PENDING$LITERAL=0 THEN RETURN;
00776 IF PENDING$LIT$IC>ID$STACK$PTR THEN RETURN;
00777 CALL START$INITIALIZE(GET$ADDRESS, HOLD:=GET$S$LENGTH);
00778 IF PENDING$LITERAL>2 THEN
00779 CC;
00780 IF PENDING$LITERAL=3 THEN CHAR='D';
00781 ELSE IF PENDING$LITERAL=4 THEN CHAR=' ';
00782 ELSE CHAR=CUTE;
00783 CALL FILL$STRING(HOLD, CHAR);
00784 END;
00785 ELSE IF PENDING$LITERAL = 2 THEN
00786 CC;
00787 IF FCLC <= FCLC$LIT THEN
00788 CALL STRING$OUT(.REST$HOLD$LIT, FCLC);
00789 ELSE CC;
00790 CALL STRING$OUT(.REST$HOLD$LIT, FCLC$LIT);
00791 CALL FILL$STRING(HOLD - (HOLD$LIT + 1), ' ');
00792 END;
00793 END;
00794 ELSE
00795 CC;
00796 /* THE NUMBER HANDLER */
00797 DECLARE (DEC, MINUS$SIGN, I, J, LIT$DEC, N$LENGTH,
00798 NUM$BEFORE, NUM$AFTER, TYPE) BYTE, ZONE LIT '10H';
00799 IF ((TYPE:=GET$TYPE)<16) OR (TYPE>20) THEN
00800 CALL PRINT$ERROR('NV');
00801 N$LENGTH=GET$S$LENGTH;
00802 DEC=GET$DECIMAL;
00803 MINUS$SIGN=FALSE;
00804 IF REST$HOLD$LIT='- ' THEN
00805 CC;
00806 MINUS$SIGN=TRUE;
00807 J=1;
00808 END;
00809 ELSE IF REST$HOLD$LIT='+' THEN J=1;
00810 ELSE J=C;
00811 LIT$DEC=0;
00812 DO I=1 TO HOLD$LIT;
00813 IF HOLD$LIT(I)='.' THEN LIT$DEC=1;
00814 END;
00815 IF LIT$DEC=0 THEN
00816 CC;
00817 NUM$BEFORE=REST$HOLD$LIT-J;
00818 NUM$AFTER=0;
00819 END;
00820 ELSE
00821 CC;
00822 NUM$BEFORE=LIT$DEC - J - 1;
00823 NUM$AFTER=REST$HOLD$LIT - LIT$DEC;
00824 END;
00825 IF (I:=N$LENGTH - DEC)<NUM$BEFORE THEN
00826 CALL PRINT$ERROR('SL');
00827 IF I>NUM$BEFORE THEN
00828 CC;
00829 I=I-NUM$BEFORE;
00830 IF MINUS$SIGN THEN
00831 CC;
00832 I=I-1;
00833 CALL BYTE$OUT('0' + ZONE);
00834 END;
00835 CALL FILL$STRING(I, '0');
00836 END;
00837 ELSE IF MINUS$SIGN THEN REST$HOLD$LIT(J)=REST$HOLD$LIT(J)+ZONE;
00838 CALL STRING$OUT(.REST$HOLD$LIT + J, NUM$BEFORE);
00839 IF NUM$AFTER > DEC THEN NUM$AFTER = DEC;
00840 CALL STRING$OUT(.REST$HOLD$LIT + LIT$DEC, NUM$AFTER);
00841 IF (I:=DEC - NUM$AFTER)<0 THEN
00842 CALL FILL$STRING(I, '0');
00843 END;
00844 PENDING$LITERAL=0;
00845 END REDEF$CR$VALUE;
00846 1
00847 2 REDUCE$STACK: PROCEDURE;
00848 DECLARE HCLC$LENGTH ADDRESS;
00849 CALL SET$CLR$SYM;
00850 CALL REDEF$CR$VALUE;
00851 HOLD$LENGTH=GET$S$LENGTH;
00852 IF GET$TYPE > 128 THEN
00853 CC;
00854 HOLD$LENGTH=HOLD$LENGTH * GET$OCCURS;
00855 END;
00856 ID$STACK$PTR=ID$STACK$PTR - 1;
00857 CALL SET$CLR$SYM;
00858 CALL SET$S$LENGTH(GET$S$LENGTH + HOLD$LENGTH);
00859 CALL SET$TYPE(GROUP);
00860 END REDUCE$STACK;

```



```

00861 1  ENC$CF$RECORD: PROCEDURE;
00862 2  DO WHILE IC$STACK$PTR<>0;
00863 2  CALL REDUCE$STACK;
00864 2  END;
00865 2  CALL SET$CLR$SYM;
00866 2  CALL RECEP$CR$VALUE;
00867 2  IC$STACK=C;
00868 2  TEMP$HOLD=ALLOCATE(TEMP$TWC:=GET$S$LENGTH);
00869 2  END  END$CF$RECORD;
00870 1
00871 1  CCNVERT$INTEGER: PROCEDURE;
00872 2  DECLARE INTEGER ADDRESS;
00873 2  INTEGER=0;
00874 2  CC I = 1 TO VARC;
00875 2  INTEGER=SHL(INTEGER,3)+SHL(INTEGER,1)+(VARC(I)-'0');
00876 2  END;
00877 2  VALUE(SP)=INTEGER;
00878 2  END CCNVERT$INTEGER;
00879 1
00880 1  CR$VALUE: PROCEDURE(PTR,ATTRIB);
00881 2  DECLARE PTR BYTE, ATTRIB ADDRESS;
00882 2  VALUE(PTR)=VALUE(PTR) OR ATTRIB;
00883 2  END CR$VALUE;
00884 1
00885 1  BUILD$FCB: PROCEDURE;
00886 2  DECLARE TEMP ADDRESS;
00887 2  DECLARE BUFFER(11) BYTE, (CHAR, I, J) BYTE;
00888 2  CALL FILL(.BUFFER,' ',11);
00889 2  J,I=0;
00890 2  DO WHILE (J < 11) AND (I < VARC);
00891 3  IF (CHAR:=VARC(I:=I+1))='.' THEN J=8;
00892 3  ELSE DO;
00893 4  BUFFER(J)=CHAR;
00894 4  J=J+1;
00895 3  END;
00896 2  END;
00897 2  CALL SET$ACCR2(TEMP:=ALLOCATE(164));
00898 2  CALL START$INITIALIZE(TEMP,16);
00899 2  CALL BYTES$CCT(0);
00900 2  CALL STRING$CUT(.BUFFER,11);
00901 2  CALL FILL$STRING(4,0);
00902 2  CALL CR$VALUE(SP-1,1);
00903 2  END BUILD$FCB;
00904 1
00905 1  SET$SIGN: PROCEDURE(NUMB);
00906 2  DECLARE NUMB BYTE;
00907 2  IF GET$TYPE=17 THEN CALL SET$TYPE(VALUE(SP) + NUMB);
00908 2  ELSE CALL PRINT$ERROR('SG');
00909 2  IF VALUE(SP)<>0 THEN CALL SET$S$LENGTH(GET$S$LENGTH + 1);
00910 2  END SET$SIGN;
00911 1
00912 1  PIC$ANALYZER: PROCEDURE;
00913 2  DECLARE /* WORK AREAS AND VARIABLES */
00914 2  FLAG BYTE,
00915 2  FIRST BYTE,
00916 2  CCOUNT ADDRESS,
00917 2  BUFFER (31) BYTE,
00918 2  SAVE BYTE,
00919 2  REPTITIONS ADDRESS,
00920 2  J BYTE,
00921 2  DEC$CCOUNT BYTE,
00922 2  CHAR BYTE,
00923 2  I BYTE,
00924 2  TEMP ADDRESS,
00925 2  TYPE BYTE,
00926 2
00927 2  /* * * MASKS * * */
00928 2  ALPHA LIT '0',
00929 2  A$ECIT LIT '2',
00930 2  A$EN LIT '4',
00931 2  ECIT LIT '8',
00932 2  NUM LIT '16',
00933 2  NUM$EDIT LIT '32',
00934 2  CEC LIT '64',
00935 2  SIGN LIT '128',
00936 2
00937 2  NUM$MASK LIT '1C101111B',
00938 2  NUM$ED$MASK LIT '10000101B',
00939 2  SIGN$MASK LIT '00101111B',
00940 2  A$E$MASK LIT '11111100B',
00941 2  A$N$MASK LIT '11101010B',
00942 2  A$N$E$MASK LIT '11100000B',
00943 2
00944 2  /* TYPES */
00945 2  A$TYPE LIT '80',
00946 2  N$TYPE LIT '16',
00947 2  S$TYPE LIT '17',
00948 2  I$TYPE LIT '8',
00949 2  A$TYPE LIT '72',
00950 2  AN$TYPE LIT '5',
00951 2  AN$TYPE LIT '73';
00952 1
00953 1  INC$COUNT: PROCEDURE(SWITCH);
00954 2  DECLARE SWITCH BYTE;
00955 2  FLAG=FLAG OR SWITCH;
00956 2  IF (CCOUNT:=COUNT + 1) < 31 THEN BUFFER(COUNT) = CHAR;
00957 2  END INC$COUNT;
00958 1
00959 1  CHECK: PROCEDURE (MASK) BYTE;
00960 2  /* THIS ROUTINE CHECKS A MASK AGAINST THE
00961 2  FLAG BYTE AND RETURNS TRUE IF THE FLAG
00962 2  HAS NO BITS IN COMMON WITH THE MASK */
00963 2  DECLARE MASK BYTE;
00964 2  RETURN NOT ( (FLAG AND MASK) <> 0);
00965 2  END CHECK;
00966 2

```



```

009667 2 PIC$ALLCCATE: PROCEDURE(AMT) ADDRESS;
009668 DECLARE AMT ADDRESS;
009669 IF (MAX$INT$MEM:=MAX$INT$MEM - AMT) < NEXT$AVAILABLE
009670 THEN CALL FATAL$ERROR ('MO');
009671 RETURN MAX$INT$MEM;
009672 END PIC$ALLCCATE;
009673
009674 /* PROCEDURE EXECUTION STARTS HERE */
009675
009676 CCOUNT, FLAG, DEC$CCOUNT=0;
009677 /* CHECK FOR EXCESSIVE LENGTH */
009678 IF VARC > 20 THEN
009679 CC:
009680 CALL PRINT$ERROR('PC');
009681 RETURN;
009682
009683 FAD:
009684 /* SET FLAG BITS AND COUNT LENGTH */
009685 I=1;
009686 CC WHILE IK=VARC;
009687 IF (CHAR:=VARC(I))='A' THEN CALL INC$CCOUNT(Alpha);
009688 ELSE IF CHAR='B' THEN CALL INC$COUNT(ASEDIT);
009689 ELSE IF CHAR='9' THEN CALL INC$COUNT(NUM);
009690 ELSE IF CHAR='X' THEN CALL INC$COUNT(ASN);
009691 ELSE IF (CHAR='S') AND (COUNT=0) THEN
009692 FLAG=FLAG OR SIGN;
009693 ELSE IF (CHAR='V') AND (DEC$CCOUNT=0) THEN
009694 DEC$CCOUNT=CCOUNT;
009695 ELSE IF (CHAR='/' OR (CHAR='0')) THEN CALL INC$CCOUNT(EDIT);
009696 ELSE IF
009697 (CHAR='Z') OR (CHAR='.') OR (CHAR='*') OR
009698 (CHAR='+') OR (CHAR='-') OR (CHAR='$') THEN
009699 CALL INC$COUNT(NUM$EDIT);
009700 ELSE IF (CHAR='.') AND (DEC$COUNT=0) THEN
009701 DC:
009702 CALL INC$COUNT(NUM$EDIT);
009703 DEC$CCOUNT=CCOUNT;
009704
009705 END;
009706 ELSE IF ((CHAR='C') AND (VARC(I+1)='R')) OR
009707 ((CHAR='D') AND (VARC(I+1)='B')) THEN
009708 DO:
009709 CALL INC$COUNT(NUM$EDIT);
009710 CHAR=VARC(I:=I+1);
009711 CALL INC$COUNT(NUM$EDIT);
009712
009713 END;
009714 ELSE IF (CHAR='(') AND (COUNT<>0) THEN
009715 DC:
009716 SAVE=VARC(I-1);
009717 REPITITIONS=0;
009718 CC WHILE (CHAR:=VARC(I:=I+1))<>' ';
009719 REPITITIONS=SHL(REPITITIONS,3) +
009720 SHL(REPITITIONS,1) + (CHAR - '0');
009721 END;
009722 CHAR=SAVE;
009723 CC J=1 TO REPITITIONS-1;
009724 CALL INC$COUNT(0);
009725 END;
009726 END;
009727 ELSE DC:
009728 CALL PRINT$ERROR('PC');
009729 RETURN;
009730
009731 EAC:
009732 I:=I+1;
009733 END; /* END OF DC WHILE IK=VARC */
009734 /* AT THIS POINT THE TYPE CAN BE DETERMINED */
009735 IF NOT CHECK(NUM$EDIT) THEN
009736 CC:
009737 IF CHECK(NUM$ED$MASK) THEN TYPE=NETYPE;
009738
009739 END;
009740 IF CHECK(NUM$MASK) THEN TYPE=NTYPE;
009741 IF CHECK(SNUM$MASK) THEN TYPE=SNSTYPE;
009742 IF CHECK(NOT(Alpha)) THEN TYPE=ATYPE;
009743 IF CHECK(ASE$MASK) THEN TYPE=AESTYPE;
009744 IF CHECK(ASN$MASK) THEN TYPE=ANETYPE;
009745 IF CHECK(ASNS$MASK) THEN TYPE=ANETYPE;
009746 IF TYPE=0 THEN CALL PRINT$ERROR('PC');
009747 ELSE
009748 CC:
009749 IF REDEF THEN CUR$SYM=REDEF$TWO;
009750 ELSE CLR$SYM = HOLD$SYM;
009751 CALL SET$TYPE(TYPE);
009752 CALL SET$LENGTH(COUNT + GET$S$LENGTH);
009753 IF (TYPE AND 64) <> 0 THEN
009754 DC:
009755 CALL SET$ADDR2(TEMP:=PIC$ALLOCATE(COUNT));
009756 CALL START$INITIALIZE(TEMP,COUNT);
009757 CALL STRING$OUT(.BUFFER + 1,COUNT);
009758
009759 END;
009760 IF DEC$COUNT<>0 THEN CALL SET$DECIMAL(COUNT-DEC$CCOUNT);
009761
009762 END;
009763 END PIC$ANALIZER;
009764
009765 SET$FILE$ATIRIB: PROCEDURE;
009766 DECLARE TEMP ADDRESS, TYPE BYTE;
009767 IF CUR$SYM<>VALUE(MPP1) THEN
009768 CC:
009769 TEMP=CLR$SYM;
009770 CLR$SYM=VALUE(MPP1);
009771 SYMBCL$ADDR(REL$ID)=TEMP;
009772
009773 END;
009774 IF NOT (TEMP:=VALUE(SP-1)) THEN CALL PRINT$ERROR ('NF');
009775 ELSE
009776 DC:
009777 IF TEMP=1 THEN TYPE=SEQUENTIAL;
009778 ELSE IF TEMP=15 THEN TYPE=RANDOM;
009779 ELSE IF TEMP=9 THEN TYPE=SEQ$RELATIVE;
009780 ELSE CC:
009781 CALL PRINT$ERROR('IA');
009782 TYPE=1;
009783
009784 END;
009785
009786 CALL SET$TYPE(TYPE);
009787 END SET$FILE$ATIRIB;

```



```

01077 1
01078 1 LCAC$LITERAL: FRCCEURE;
01079 1 DECLARE I BYTE;
01080 1 IF PENDING$LITERAL <> 0 THEN CALL PRINT$ERROR ('LE');
01081 1 ELSE CC I = 0 TO VARC;
01082 1 HCLD$LIT(I)=VARC(I);
01083 1 END;
01084 1 END LCAC$LITERAL;
01085 1
01086 1
01087 1 CHECK$FCR$LEVEL: PROCEDURE;
01088 1 DECLARE NEW$LEVEL BYTE;
01089 1 HOLD$SYM,CUR$SYM=VALUE(MP-1);
01090 1 CALL SET$LEVEL(NEW$LEVEL:=VALUE(MP-2));
01091 1 IF NEW$LEVEL=1 THEN
01092 1 CO;
01093 1 IF 1C$STACK<>0 THEN
01094 1 CC;
01095 1 IF NOT FILE$SEC&END THEN
01096 1 CO;
01097 1 CALL SET$REDEF(ID$STACK,VALUE(MP-1));
01098 1 VALUE(MP)=1; /* SET REDEFINE FLAG */
01099 1 END;
01100 1 CALL END$OF$RECORD;
01101 1 END;
01102 1 ELSE CC FILE $STACK$LEVEL >= NEW$LEVEL;
01103 1 CALL REDUCE$STACK;
01104 1 END;
01105 1 END CHECK$FCR$LEVEL;
01106 1
01107 1
01108 1 CCDE$GEN: FFOCECLRE(PRODUCTION);
01109 1 DECLARE PRODUCTION BYTE;
01110 1 IF PRINT$PRCC THEN
01111 1 CO;
01112 1 CALL CRLF;
01113 1 CALL PRINTCHAR(POUND);
01114 1 CALL PRINT$NUMBER(PRODUCTION);
01115 1 END;
01116 1
01117 1 CC CASE PRODUCTION;
01118 1
01119 1 /* P R O D U C T I O N S */
01120 1
01121 1 /* CASE 0 NOT USED */
01122 1
01123 1 /*
01124 1 1 <PROGRAM> ::= <ID-DIV> <E-DIV> <C-DIV> PROCEDURE
01125 1 CCMPILING=FALSE;
01126 1 2 <ID-DIV> ::= IDENTIFICATION DIVISION . PROGRAM-ID .
01127 1 /* <COMMENT> . <AUTH> <CATE> <SEC>
01128 1 ; /* NO ACTION REQUIRED */
01129 1 3 <AUTH> ::= AUTHOR . <COMMENT> .
01130 1 ; /* NO ACTION REQUIRED */
01131 1 4 <EMPTY>
01132 1 ; /* NO ACTION REQUIRED */
01133 1 5 <CATE> ::= DATE-WRITTEN . <COMMENT> .
01134 1 ; /* NO ACTION REQUIRED */
01135 1 6 <EMPTY>
01136 1 ; /* NO ACTION REQUIRED */
01137 1 7 <SEC> ::= SECURITY . <COMMENT> .
01138 1 ; /* NO ACTION REQUIRED */
01139 1 8 <EMPTY>
01140 1 ; /* NO ACTION REQUIRED */
01141 1 9 <COMMENT> ::= <INPUT>
01142 1 ; /* NO ACTION REQUIRED */
01143 1 10 <COMMENT> <INPUT>
01144 1 ; /* NO ACTION REQUIRED */
01145 1 11 <E-DIV> ::= ENVIRONMENT DIVISION . CONFIGURATION
01146 1 /* SECTION . <SRC-CBJ> <I-O>
01147 1 ; /* NO ACTION REQUIRED */
01148 1 12 <SRC-CBJ> ::= SOURCE-COMPUTER . <COMMENT> <DEBUG> .
01149 1 /* OBJECT-COMPUTER . <COMMENT> .
01150 1 ; /* NO ACTION REQUIRED */
01151 1 13 <DEBUG> ::= DEBUGGING MODE
01152 1 /* DEBUGGING=TRUE; /* SETS A SCANNER TOGGLE */
01153 1 14 <EMPTY>
01154 1 ; /* NO ACTION REQUIRED */
01155 1 15 <I-O> ::= INPUT-OUTPUT SECTION . FILE-CONTROL .
01156 1 /* <FILE-CONTROL-LIST> <IC>
01157 1 ; /* NO ACTION REQUIRED */
01158 1 16 <EMPTY>
01159 1 ; /* NO ACTION REQUIRED */
01160 1 17 <FILE-CONTROL-LIST> ::= <FILE-CONTROL-ENTRY>
01161 1 ; /* NO ACTION REQUIRED */
01162 1 18 <FILE-CONTROL-LIST>
01163 1 /* <FILE-CONTROL-ENTRY>
01164 1 ; /* NO ACTION REQUIRED */
01165 1 19 <FILE-CONTROL-ENTRY> ::= SELECT <ID> <ATTRIBUTE-LIST> .
01166 1 /* CALL SET$FILE$ATTRIB;
01167 1 20 <ATTRIBUTE-LIST> ::= <ONE-ATTRIB>
01168 1 ; /* NO ACTION REQUIRED */
01169 1 21 <ATTRIBUTE-LIST> <ONE-ATTRIB>
01170 1 /* VALUE(MP)=VALUE(SP) OR VALUE(MP);
01171 1 22 <ONE-ATTRIB> ::= ORGANIZATION <ORG-TYPE>
01172 1 /* VALUE(MP)=VALUE(SP);
01173 1 23 ACCESS <ACC-TYPE> <RELATIVE>
01174 1 /* VALUE(MP)=VALUE(MP+1) OR VALUE(SP);
01175 1 24 ASSIGN <INPUT>
01176 1 /* CALL BUILD$FCB;
01177 1 25 <ORG-TYPE> ::= SEQUENTIAL
01178 1 ; /* NO ACTION REQUIRED - DEFAULT */
01179 1 26 RELATIVE
01180 1 /* CALL CR$VALUE(SP,4);
01181 1 27 <ACC-TYPE> ::= SEQUENTIAL
01182 1 ; /* NO ACTION REQUIRED - DEFAULT */
01183 1 28 RANDOM
01184 1 /* CALL CR$VALUE(SP,2);
01185 1 29 <RELATIVE> ::= RELATIVE <ID>
01186 1 /* CALL CR$VALUE(MP,8);

```



```

01187      /*      30      <EMPTY>                                     */
01188      /* ;      /* NO ACTION REQUIRED - DEFAULT */                      */
01189      /* ;      31      <IC> ::= 1-0-CONTROL . <SAME-LIST>          */
01190      /* ;                                     <EMPTY>                */
01191      /* ;      32      <SAME-LIST> ::= <SAME-ELEMENT>                */
01192      /* ;      33      <SAME-ELEMENT> ::= <SAME-LIST> <SAME-ELEMENT> */
01193      /* ;      34      <SAME-ELEMENT> ::= SAME <ID-STRING> .        */
01194      /* ;      35      <ID-STRING> ::= <ID>                          */
01195      /* ;      36      <ID-STRING> ::= <ID>                          */
01196      /* ;      37      <ID-STRING> <ID>                             */
01197      /* ;                                     <EMPTY>                */
01198      /* ;      38      <C-DIV> ::= DATA DIVISION . <FILE-SECTION> <WORK> */
01199      /* ;      39      <FILE-SECTION> ::= FILE SECTION . <FILE-LIST> */
01200      /* ;      40      FILE$SECT$END = TRUE;                          */
01201      /* ;      41      <FILE-LIST> ::= <FILES>                       */
01202      /* ;      42      <FILES> ::= <FILE-LIST> <FILES>                */
01203      /* ;      43      <FILES> ::= FD <ID> <FILE-CONTROL> .          */
01204      /* ;      44      <FILE-CONTROL> ::= <FILE-LIST>                */
01205      /* ;      45      <FILE-ELEMENT> ::= BLOCK <INTEGER> RECORDS,  */
01206      /* ;      46      <FILE-ELEMENT> ::= RECORD <REC-COUNT>        */
01207      /* ;      47      CALL SET$LENGTH(VALUE($P));                  */
01208      /* ;      48      LABEL RECORDS STANDARD                        */
01209      /* ;      49      LABEL RECORDS OMITTED                         */
01210      /* ;      50      VALUE OF <ID-STRING>                          */
01211      /* ;      51      <REC-COUNT> ::= <INTEGER>                      */
01212      /* ;      52      <REC-COUNT> ::= VALUE($P) CORRECT */
01213      /* ;      53      <REC-COUNT> TO <INTEGER>                      */
01214      /* ;      54      CC;                                           */
01215      /* ;      VALUE($P)=VALUE($P); /* VARIABLE LENGTH */          */
01216      /* ;      CALL SET$TYPE(4); /* SET TO VARIABLE */              */
01217      /* ;      END;                                                 */
01218      /* ;      55      <WORK> ::= WORKING-STORAGE SECTION .          */
01219      /* ;      56      <RECORD-DESCRIPTION>                          */
01220      /* ;      57      <LINK> ::= LINKAGE SECTION . <RECORD-DESCRIPTION> */
01221      /* ;      58      CALL PRINT$ERROR('N'); /* INTER PROG COMM */ */
01222      /* ;      59      <RECORD-DESCRIPTION> ::= <LEVEL-ENTRY>        */
01223      /* ;      60      <RECORD-DESCRIPTION> <RECORD-DESCRIPTION>    */
01224      /* ;      61      <LEVEL-ENTRY> ::= <LEVEL-ENTRY>                */
01225      /* ;      62      <LEVEL-ENTRY> ::= <INTEGER> <DATA-ID> <REDEFINES> */
01226      /* ;      63      <LEVEL-ENTRY> ::= <DATA-TYPE> .              */
01227      /* ;      CC;                                                 */
01228      /* ;      CALL LEAD$LEVEL;                                       */
01229      /* ;      IF PENDING$LITERAL<>0 THEN PENDING$LIT=ID=ID$STACK$PTR; */
01230      /* ;      END;                                                 */
01231      /* ;      64      <DATA-ID> ::= <ID>                             */
01232      /* ;      65      <DATA-ID> ::= FILLER                           */
01233      /* ;      CC;                                                 */
01234      /* ;      CLR$SYM, VALUE($P)=NEXT$SYM;                          */
01235      /* ;      CALL BLI0$SYMBOL(0);                                  */
01236      /* ;      END;                                                 */
01237      /* ;      66      <REDEFINES> ::= REDEFINES <ID>                */
01238      /* ;      CC;                                                 */
01239      /* ;      CALL SET$REDEF(VALUE($P),VALUE($P-2));                */
01240      /* ;      VALUE($P)=1; /* SET REDEFINE FLAG ON */              */
01241      /* ;      CALL CHECK$FOR$LEVEL;                                  */
01242      /* ;      END;                                                 */
01243      /* ;      67      <CHECK$FOR$LEVEL> ::= <EMPTY>                */
01244      /* ;      68      <DATA-TYPE> ::= <PROP-LIST>                    */
01245      /* ;      69      <DATA-TYPE> ::= <EMPTY>                        */
01246      /* ;      70      <PROP-LIST> ::= <DATA-ELEMENT>                */
01247      /* ;      71      <PROP-LIST> ::= <PROP-LIST> <DATA-ELEMENT>    */
01248      /* ;      72      <DATA-ELEMENT> ::= PIC <INPUT>                */
01249      /* ;      CALL PIC$ANALYZER;                                     */
01250      /* ;      CALL SET$TYPE(COMP);                                   */
01251      /* ;      73      USAGE COMP                                     */
01252      /* ;      74      USAGE DISPLAY                                  */
01253      /* ;      75      /* NO ACTION REQUIRED - DEFAULT */              */

```



```

01297 3      /* 73 SIGN LEADING <SEPARATE> */
01298 3333 /* CALL SET$SIGN(18); SIGN TRAILING <SEPARATE> */
01299 3333 /* 74 SIGN TRAILING <SEPARATE> */
01300 3333 /* CALL SET$SIGN(17); OCCURS <INTEGER> */
01301 3333 /* 75 OCCURS <INTEGER> */
01302 3333 CC;
01303 3333 CALL CR$TYPE(128);
01304 3333 CALL SET$OCCURS(VALUE(SP));
01305 3333 END;
01306 3333 /* 76 SYNC <DIRECTION> */
01307 3333 /* ; /* NG ACTION REQUIRED - BYTE MACHINE */
01308 3333 /* 77 VALUE <LITERAL> */
01309 3333 /* CC;
01310 3333 IF NOT FILE$SECSEND THEN
01311 3333 DC;
01312 3333 CALL PRINT$ERROR('VE');
01313 3333 PENDING$LITERAL=0;
01314 3333 END;
01315 3333 END;
01316 3333 /* 78 <DIRECTION> ::= LEFT */
01317 3333 /* ; /* NO ACTION REQUIRED */
01318 3333 /* 79 RIGHT */
01319 3333 /* ; /* NO ACTION REQUIRED */
01320 3333 /* 80 <EMPTY> */
01321 3333 /* ; /* NO ACTION REQUIRED */
01322 3333 /* 81 <SEPARATE> ::= SEPARATE */
01323 3333 /* VALUE(SF)=2; */
01324 3333 /* 82 <EMPTY> */
01325 3333 /* ; /* NO ACTION REQUIRED */
01326 3333 /* 83 <LITERAL> ::= <INPUT> */
01327 3333 /* CC;
01328 3333 CALL LCAD$LITERAL;
01329 3333 PENDING$LITERAL=1;
01330 3333 END;
01331 3333 /* 84 <LIT> */
01332 3333 /* CC;
01333 3333 CALL LCAD$LITERAL;
01334 3333 PENDING$LITERAL=2;
01335 3333 END;
01336 3333 /* 85 ZERC */
01337 3333 /* PENDING$LITERAL=3; */
01338 3333 /* 86 SPACE */
01339 3333 /* PENDING$LITERAL=4; */
01340 3333 /* 87 QUOTE */
01341 3333 /* PENDING$LITERAL=5; */
01342 3333 /* 88 <INTEGER> ::= <INPUT> */
01343 3333 /* CALL CONVERT$INTEGER; */
01344 3333 /* 89 <IC> ::= <INPUT> */
01345 3333 /* VALUE(SP)=MATCH; /* STORE SYMBOL TABLE FCINTERS */
01346 3333
01347 3333
01348 3333 END; /* ENL OF CASE STATEMENT */
01349 3333 END CCCE$GEN;
01350 3333
01351 3333 GETIN1: PROCEDURE BYTE;
01352 3333 RETURN INDEX1(STATE);
01353 3333 END GETIN1;
01354 3333
01355 3333 GETIN2: PROCEDURE BYTE;
01356 3333 RETURN INDEX2(STATE);
01357 3333 END GETIN2;
01358 3333
01359 3333 INCSF: PROCEDURE;
01360 3333 SF=SP+1;
01361 3333 IF SP >= PSTACKSIZE THEN CALL FATAL$ERROR('SO');
01362 3333 VALUE(SF)=0; /* CLEAR VALUE STACK */
01363 3333 END INCSF;
01364 3333
01365 3333 LCCKAHEAD: PROCEDURE;
01366 3333 IF NOT LCCK THEN
01367 3333 CC;
01368 3333 CALL SCANNER;
01369 3333 NOTLCCK=FALSE;
01370 3333 IF PRINT$TOKEN THEN
01371 3333 CC;
01372 3333 CALL CRLF;
01373 3333 CALL PRINT$NUMBER(TOKEN);
01374 3333 CALL PRINT$CHAR(' ');
01375 3333 CALL PRINT$ACCUM;
01376 3333 END;
01377 3333 END LCCKAHEAD;
01378 3333
01379 3333 NC$CCFLICT: PROCEDURE (CSTATE) BYTE;
01380 3333 DECLARE (CSTATE,1,J,K) BYTE;
01381 3333 J=INDEX1(CSTATE);
01382 3333 K=J+INDEX2(CSTATE)-1;
01383 3333 CC I=J TO K;
01384 3333 IF READ1(I)=TOKEN THEN RETURN TRUE;
01385 3333 END;
01386 3333
01387 3333 RETL$N FALSE;
01388 3333 END NC$CCFLICT;
01389 3333
01390 3333 RECOVER: PROCEDURE BYTE;
01391 3333 DECLARE (TSP, RSTATE) BYTE;
01392 3333 CC FOREVER;
01393 3333 TSP=SF;
01394 3333 DO WHILE TSP <> 255;
01395 3333 IF NOT NC$CCFLICT(RSTATE:=STATESTACK(TSP)) THEN
01396 3333 CC; /* STATE WILL READ TOKEN */
01397 3333 IF SP <> TSP THEN SP = TSP - 1;
01398 3333 RETURN RSTATE;
01399 3333 END;
01400 3333 TSP = TSP - 1;
01401 3333 END;
01402 3333 CALL SCANNER; /* TRY ANOTHER TOKEN */
01403 3333 END;
01404 3333 END RECOVER;

```



```

01405 1
01406 1
01407 1
01408 1
01409 1
01410 1
01411 1
01412 1
01413 1
01414 1
01415 1
01416 1
01417 1
01418 1
01419 1
01420 1
01421 1
01422 1
01423 1
01424 1
01425 1
01426 1
01427 1
01428 1
01429 1
01430 1
01431 1
01432 1
01433 1
01434 1
01435 1
01436 1
01437 1
01438 1
01439 1
01440 1
01441 1
01442 1
01443 1
01444 1
01445 1
01446 1
01447 1
01448 1
01449 1
01450 1
01451 1
01452 1
01453 1
01454 1
01455 1
01456 1
01457 1
01458 1
01459 1
01460 1
01461 1
01462 1
01463 1
01464 1
01465 1
01466 1
01467 1
01468 1
01469 1
01470 1
01471 1
01472 1
01473 1
01474 1
01475 1
01476 1
01477 1
01478 1
01479 1
01480 1
01481 1
01482 1
01483 1
01484 1
01485 1
01486 1
01487 1
01488 1
01489 1
01490 1
01491 1
01492 1
01493 1
01494 1
01495 1
01496 1
01497 1
01498 1
01499 1
01500 1
01501 1
01502 1

END$PASS: PROCEDURE;
/* THIS PROCEDURE STORES THE INFORMATION REQUIRED BY PASS2
IN LOCATIONS ABOVE THE SYMBOL TABLE. THE FOLLOWING
INFORMATION IS STORED:
OUTPUT FILE CONTROL BLOCK
COMPILER TOGGLES
INFLUT BUFFER POINTER
THE OUTFLUT BUFFER IS ALSO FILLED SO THE CURRENT RECORD IS WRITTEN.
*/
CALL BYTE$CLT(SCD);
CALL ADDR$CLT(NEXT$AVAILABLE);
CC WHILE CLT$PTR<>.OUTPUT$BUFF;
CALL EYTE$OUT(OFH);
END;

CALL MOVE(.OUTPUT$FCB,MAX$MEMCRY-PASS1$LEN,PASS1$LEN);
GO TO MAX$MEMORY;
END END$PASS;

/* * * * * PROGRAM EXECUTION STARTS HERE * * */

CALL MOVE(INITIAL$POS,MAX$MEMCRY,RDR$LENGTH);
CALL INIT$SCANNER;
CALL INIT$SYMBOL;

/* * * * * * * * * * * * * * * * * * * * * */
DC WHILE COMPILING;
IF STATE <= MAX$RND THEN /* READ STATE */
CC;
CALL INCSP;
STATE$STACK(SP) = STATE; /* SAVE CURRENT STATE */
CALL LCKAHEAD;
I=GETIN1;
J = I + GETIN2 - 1;
CC I=I TO J;
IF READ1(I) = TOKEN THEN
CC;
/* COPY THE ACCUMULATOR IF IT IS AN INPUT
STRING. IF IT IS A RESERVED WORD IT DOES
NOT NEED TO BE COPIED. */
IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
DO K=0 TO ACCUM;
VARC(K)=ACCUM(K);
END;
STATE=READ2(I);
NOLOCK=TRUE;
I=J;
END;
ELSE
IF I=J THEN
CC;
CALL PRINT$ERR$FOR('NP');
CALL PRINT(' ERROR NEAR $');
CALL PRINT$ACCUM;
IF (STATE:=RECOVER)=0 THEN COMPILING=FALSE;
END;
END; /* END OF READ STATE */
ELSE
IF STATE>MAX$PND THEN /* APPLY PRODUCTION STATE */
CC;
MP=SP - GETIN2;
MPP1=MP + 1;
CALL CCC$GEN(STATE - MAX$PND);
SP=MP;
I=GETIN1;
J=STATE$STACK(SP);
DO WHILE (K:=APPLY1(I)) <> 0 AND J<>K;
I=I + 1;
END;
IF (K:=APPLY2(I))=0 THEN COMPILING=FALSE;
STATE=K;
END;
ELSE
IF STATE<=MAX$LNC THEN /*LOOKAHEAD STATE*/
CC;
I=GETIN1;
CALL LCKAHEAD;
CC WHILE (K:=LOOK1(I))<>0 AND TOKEN <>K;
I=I+1;
END;
STATE=LCK2(I);
END;
ELSE
CC; /*PUSH STATES*/
CALL INCSP;
STATE$STACK(SP)=GETIN2;
STATE=GETIN1;
END;
END; /* CF WHILE COMPILING */
CALL CRLF;
CALL PRINT('END OF PART 1 $');
CALL END$PASS;
ECF

```


[illegible]


```

00106 1
00107 1
00108 1
00109 1
00110 1
00111 1
00112 1
00113 1
00114 1
00115 1
00116 2
00117 2
00118 2
00119 1
00120 1
00121 1
00122 2
00123 2
00124 1
00125 1
00126 2
00127 2
00128 2
00129 2
00130 1
00131 1
00132 2
00133 2
00134 1
00135 1
00136 2
00137 2
00138 2
00139 2
00140 1
00141 2
00142 2
00143 2
00144 2
00145 2
00146 1
00147 1
00148 2
00149 2
00150 2
00151 2
00152 2
00153 1
00154 1
00155 2
00156 2
00157 1
00158 2
00159 2
00160 2
00161 1
00162 2
00163 2
00164 2
00165 1
00166 1
00167 2
00168 2
00169 2
00170 2
00171 2
00172 2
00173 1
00174 1
00175 2
00176 2
00177 2
00178 2
00179 2
00180 2
00181 2
00182 1
00183 2
00184 1
00185 1
00186 2
00187 2
00188 2
00189 2
00190 2
00191 2
00192 2
00193 2
00194 1
00195 1
00196 1
00197 1
00198 1
00199 1
00200 1
00201 1
00202 1
00203 1
00204 1
00205 1
00206 1
00207 1
00208 1
00209 1
00210 1

/* GLOBAL COUNTERS */
DECLARE
CTR BYTE,
A:CTR ADDRESS,
BASE ADDRESS,
B:BYTE BASEL BASE BYTE,
B:ADDR BASEL BASE ADDRESS;

MCN1: PROCEDURE (F,A);
DECLARE F BYTE, A ADDRESS;
GC TO BCCS;
END MCN1;

MCN2: PROCEDURE (F,A) BYTE;
DECLARE F BYTE, A ADDRESS;
GC TO BCCS;
END MCN2;

PRINTCHAR: PROCEDURE (CHAR);
DECLARE CHAR BYTE;
CALL MCN1 (2,CHAR);
END PRINTCHAR;

CRLF: PROCEDURE;
CALL PRINTCHAR(CR);
CALL PRINTCHAR(LF);
END CRLF;

PRINT: PROCEDURE (A);
DECLARE A ADDRESS;
CALL MCN1 (9,A);
END PRINT;

PRINT$ERROR: PROCEDURE (CODE);
DECLARE CODE ADDRESS;
CALL CRLF;
CALL PRINTCHAR(HIGH(CODE));
CALL PRINTCHAR(LOW(CODE));
END PRINT$ERROR;

FATAL$ERROR: PROCEDURE (REASON);
DECLARE REASON ADDRESS;
CALL PRINT$ERROR(REASON);
CALL TIME(1);
GC TO BCCS;
END FATAL$ERROR;

CLOSE: PROCEDURE;
IF MCN2(16,CUTPUT$FCB)=255 THEN CALL FATAL$ERROR('CL');
END CLOSE;

MCRE$INPUT: PROCEDURE BYTE;
/* READS THE INPUT FILE AND RETURNS TRUE IF A RECORD
WAS READ. FALSE IMPLIES END OF FILE */
DECLARE CCNT BYTE;
IF (CCNT:=MCN2(20,INPUT$FCB))>1 THEN CALL FATAL$ERROR('BR');
RETURN NOT(CCNT);
END MCRE$INPUT;

WRITE$OUTPUT: PROCEDURE (LOCATION);
/* WRITES CNT A 128 BYTE BUFFER FROM LOCATION*/
DECLARE LOCATION ADDRESS;
CALL MCN1(24,LOCATION); /* SET DMA */
IF MCN2(21,CUTPUT$FCB)<0 THEN CALL FATAL$ERROR('WR');
CALL MCN1(24,80H); /*RESET DMA */
END WRITE$OUTPUT;

MOVE: PROCEDURE(SOURCE, DESTINATION, COUNT);
/* MOVES FOR THE NUMBER OF BYTES SPECIFIED BY COUNT */
DECLARE (SOURCE,DESTINATION) ADDRESS;
C$BYTE BASED SOURCE, D$BYTE BASED DESTINATION, COUNT) BYTE;
DC WHILE (CCNT:=COUNT - 1)<>255;
D$BYTE:=S$BYTE;
SOURCE=SOURCE+1;
DESTINATION = DESTINATION + 1;
END;
END MOVE;

FILL: PROCEDURE(ADDR,CHAR,COUNT);
/* MOVES CHAR INTO ADDR FOR COUNT BYTES */
DECLARE ADDR ADDRESS;
(CHAR,CCLNT,CST BASED ADDR) BYTE;
DC WHILE (CCLNT:=COUNT -1)<>255;
D$1=CHAR;
ADDR=ADDR + 1;
END;
END FILL;

/* * * * * * SCANNER LITS * * * * */
DECLARE
LITERAL LIT '28',
INPUT$STP LIT '47',
PERIOD LIT '1',
RPAPIN LIT '3',
LPAPIN LIT '2',
INVALID LIT '0';

/* * * * * * SCANNER TABLES * * * * */
DECLARE TOKEN$TABLE DATA
/* CONTAINS THE TOKEN NUMBER ONE LESS THAN THE FIRST RESERVED WORD
FOR EACH LENGTH OF WORD */
(0,0,3,7,12,28,40,47,55,59,62),

```



```

000211 1 TABLE DATA('BY','GO','IF','TO','ADD','END','I-C'
000212 1 , 'NOT','RUN','CALL','ELSE','EXIT','FROM','INTO','LESS','MOVE'
000213 1 , 'NEXT','OPEN','PAGE','HEAD','SIZE','STOP','THRU','ZERO'
000214 1 , 'AFTER','CLOSE','ENTER','EQUAL','ERROR','INPUT','CLUTE','SPACE'
000215 1 , 'TIMES','CATIL','USING','WRITE','ACCEPT','BEFORE','DELETE'
000216 1 , 'CIVIDE','CLPUT','DISPLAY','GREATER'
000217 1 , 'INVALID','NUMERIC','PERFORM','REWRITE','PCUNDED','SECTION'
000218 1 , 'CIVISION','MULTIPLY','SENTENCE','SUBTRACT','ADVANCING',
000219 1 , 'DEPENDING','PROCEDURE','ALPHABETIC'),
000220 1 OFFSET (11) ADDRESS INITIAL
000221 1 /* NUMBER OF BYTES TO INDEX INTO THE TABLE FOR EACH LENGTH */
000222 1 (0,0,0,8,23,83,143,173,229,261,288),
000223 1
000224 1 WORDSCOUNT DATA
000225 1 /* NUMBER OF WORDS OF EACH SIZE */
000226 1 (0,0,4,5,15,12,5,8,4,3,1),
000227 1
000228 1
000229 1 MAX$ID$LEN LIT '12',
000230 1 MAX$LEN LIT '10',
000231 1 AC$END DATA ('END. '),
000232 1 LCKED BYTE INITIAL (0),
000233 1 FLD BYTE,
000234 1 BUFFER$END ADDRESS INITIAL (100H),
000235 1 NEXT BASED POINTER BYTE,
000236 1 INBUFF LIT '80H',
000237 1 CHAR BYTE INITIAL(' '),
000238 1 ACCUM BYTE,
000239 1 P$ACCUM (30) BYTE,
000240 1 DISPLAY BYTE INITIAL (0),
000241 1 DISPLAY$REST (73) BYTE,
000242 1 TCKEN BYTE; /*RETURNED FROM SCANNER */
000243 1
000244 1 /* PROCEDURES USED BY THE SCANNER */
000245 1
000246 1 NEXT$CHAR: PROCEDURE BYTE;
000247 1 IF LOCKED THEN
000248 1 CC;
000249 1 , LCKED=FALSE;
000250 1 RETRN (CHAR:=HOLD);
000251 1 END;
000252 1 IF (POINTER:=POINTER + 1) >= BUFFER$END THEN
000253 1 CC;
000254 1 IF NOT PCRE$INPUT THEN
000255 1 CC;
000256 1 BUFFER$END=.MEMORY;
000257 1 PCINTER=.AC$END;
000258 1 END;
000259 1 ELSE PCINTER=INBUFF;
000260 1 END;
000261 1 RETURN (CHAR:=NEXT);
000262 1 END NEXT$CHAR;
000263 1
000264 1 GET$CHAR: PROCEDURE;
000265 1 /* THIS PROCEDURE IS CALLED WHEN A NEW CHAR IS NEEDED WITHOUT
000266 1 THE DIRECT RETURN OF THE CHARACTER*/
000267 1 CHAR=NEXT$CHAR;
000268 1 END GET$CHAR;
000269 1
000270 1
000271 1 DISPLAY$LINE: PROCEDURE;
000272 1 IF NOT LIST$INPUT THEN RETURN;
000273 1 DISPLAY(DISPLAY + 1) = '5';
000274 1 CALL PRINT(.DISPLAY$REST);
000275 1 DISPLAY=C;
000276 1 END DISPLAY$LINE;
000277 1
000278 1 LOAD$DISPLAY: PROCEDURE;
000279 1 IF DISPLAY<72 THEN
000280 1 DISPLAY(DISPLAY:=DISPLAY+1)=CHAR;
000281 1 CALL GET$CHAR;
000282 1 END LOAD$DISPLAY;
000283 1
000284 1
000285 1 PLT: PROCEDURE;
000286 1 IF ACCUM < 20 THEN
000287 1 ACCUM(ACCUM:=ACCUM+1)=CHAR;
000288 1 CALL LOAD$DISPLAY;
000289 1 END PLT;
000290 1
000291 1 EAT$LINE: PROCEDURE;
000292 1 CC WHILE CHAR<>CR;
000293 1 CALL LOAD$DISPLAY;
000294 1 END EAT$LINE;
000295 1
000296 1 GET$NC$BLANK: PROCEDURE;
000297 1 CCLEAR (N,1) BYTE;
000298 1 CC FOREVER;
000299 1 IF CHAR = ' ' THEN CALL LOAD$DISPLAY;
000300 1 ELSE
000301 1 IF CHAR=CR THEN
000302 1 CC;
000303 1 CALL DISPLAY$LINE;
000304 1 IF SEQ$NUM THEN N=8; ELSE N=2;
000305 1 CC I = 1 TO N;
000306 1 CALL LOAD$DISPLAY;
000307 1 END;
000308 1 IF CHAR = '*' THEN CALL EAT$LINE;
000309 1 END;
000310 1 ELSE
000311 1 IF CHAR = ':' THEN
000312 1 CC;
000313 1 IF NOT DEBUGGING THEN CALL EAT$LINE;
000314 1 ELSE
000315 1 CALL LOAD$DISPLAY;
000316 1 END;
000317 1 ELSE
000318 1 RETRN;
000319 1 END; /* END OF CC FOREVER */
000320 1 END GET$NC$BLANK;

```



```

00322 I SPACE: PROCEDURE BYTE;
00323 I RETURN (CHAR=' ') CR (CHAR=CR);
00324 I END SPACE;
00325 I
00326 I LEFT$PARIN: PROCEDURE BYTE;
00327 I RETURN CHAR = '(';
00328 I END LEFT$PARIN;
00329 I
00330 I RIGHT$PARIN: PROCEDURE BYTE;
00331 I RETURN CHAR = ')';
00332 I END RIGHT$PARIN;
00333 I
00334 I DELIMITER: PROCEDURE BYTE;
00335 I /* CHECKS FOR A PERIOD FOLLOWED BY A SPACE OR CR */
00336 I IF CHAR <> '.' THEN RETURN FALSE;
00337 I MCLO=NEXT$CHAR;
00338 I LCKED=TRUE;
00339 I IF SPACE THEN
00340 I DC;
00341 I CHAR = '.';
00342 I RETURN TRUE;
00343 I
00344 I END;
00345 I CHAR='.';
00346 I RETURN FALSE;
00347 I END DELIMITER;
00348 I
00349 I END$CF$TOKEN: PROCEDURE BYTE;
00350 I RETURN SPACE OR DELIMITER OR LEFT$PARIN OR RIGHT$PARIN;
00351 I END END$CF$TOKEN;
00352 I
00353 I GET$LITERAL: PROCEDURE BYTE;
00354 I DO FOREVER;
00355 I IF NEXT$CHAR= QUOTE THEN RETURN LITERAL;
00356 I CALL PLT;
00357 I END;
00358 I END GET$LITERAL;
00359 I
00360 I LCK$UP: PROCEDURE BYTE;
00361 I DECLARE PCINT ADDRESS;
00362 I (HERE BASED PCINT,I) BYTE;
00363 I
00364 I MATCH: PROCEDURE BYTE;
00365 I DECLARE J BYTE;
00366 I DO J=1 TO ACCUM;
00367 I IF HERE(J - 1) <> ACCUM(J) THEN RETURN FALSE;
00368 I END;
00369 I RETURN TRUE;
00370 I END MATCH;
00371 I
00372 I PCINT=OFFSET(ACCUM)+.TABLE;
00373 I DO I=1 TO WCFC$COUNT(ACCUM);
00374 I IF MATCH THEN RETURN I;
00375 I PCINT = PCINT + ACCUM;
00376 I END;
00377 I RETURN FALSE;
00378 I END LCK$UP;
00379 I
00380 I RESERVED$WORD: PROCEDURE BYTE;
00381 I /* RETURNS THE TOKEN NUMBER OF A RESERVED WORD IF THE CONTENTS OF
00382 I THE ACCUMULATOR IS A RESERVED WORD, OTHERWISE RETURNS ZERO */
00383 I DECLARE VAL$ BYTE;
00384 I DECLARE NUMB BYTE;
00385 I IF ACCUM <= MAX$LEN THEN
00386 I DC;
00387 I IF (NL$B:=TOKEN$TABLE(ACCUM))<>0 THEN
00388 I DC;
00389 I IF (VALUE:=LOOK$UP) <> 0 THEN
00390 I NUMB=NUMB + VALUE;
00391 I ELSE NUMB=0;
00392 I END;
00393 I END;
00394 I RETURN NLMB;
00395 I END RESERVED$WORD;
00396 I
00397 I GET$TOKEN: PROCEDURE BYTE;
00398 I ACCUM=0;
00399 I CALL GET$INC$BLANK;
00400 I IF CHAR=QUOTE THEN RETURN GET$LITERAL;
00401 I IF DELIMITER THEN
00402 I DC;
00403 I CALL FLT;
00404 I RETURN PERIOD;
00405 I END;
00406 I IF LEFT$PARIN THEN
00407 I DC;
00408 I CALL FLT;
00409 I RETURN L$PARIN;
00410 I END;
00411 I IF RIGHT$PARIN THEN
00412 I DC;
00413 I CALL FLT;
00414 I RETURN R$PARIN;
00415 I END;
00416 I DO FOREVER;
00417 I CALL FLT;
00418 I IF END$CF$TOKEN THEN RETURN INPUT$STR;
00419 I END; /* CF DO FOREVER */
00420 I END GET$TOKEN;
00421 I
00422 I /* END OF SCANNER ROUTINES */
00423 I
00424 I /* SCANNER EXEC */
00425 I
00426 I SCANNER: PROCEDURE;
00427 I IF (TOKEN:=GET$TOKEN) = INPUT$STR THEN
00428 I IF (CTR:=RESERVED$WORD) <> 0 THEN TOKEN=CTR;
00429 I END SCANNER;
00430 I
00431 I
00432 I PRINT$ACCUM: PROCEDURE;
00433 I ACCUM(ACCUM+1)=' ';
00434 I CALL PRINT(.F$ACCUM);
00435 I END PRINT$ACCUM;

```



```

00437 1 PRINT$NUMBER: PRCEDURE(NUMB);
00438 1 DECLARE(NUMB,I,CNT,K) BYTE, J DATA(100,10);
00439 1 CC I=0; IC 1;
00440 1 CNT=0;
00441 1 DC WHILE NUMB >= (K:=J(I));
00442 1     NUMB=NUMB - K;
00443 1     CNT=CNT + 1;
00444 1 END;
00445 1 CALL PRINTCHAR('0' + CNT);
00446 1 END;
00447 1 CALL PRINTCHAR('0' + NUMB);
00448 1 END PRINT$NUMBER;
00449 1
00450 1
00451 1
00452 1 /* * * * END OF SCANNER PROCEDURES * * * */
00453 1
00454 1
00455 1 /* * * * * SYMBOL TABLE DECLARATIONS * * * */
00456 1
00457 1 DECLARE
00458 1 CUR$SYM ADDRESS, /*SYMBOL BEING ACCESSED*/
00459 1 SYMBOL BASED CUR$SYM BYTE,
00460 1 SYMBOL$ADDR BASED CUR$SYM ADDRESS,
00461 1 NEXT$SYM$ENTRY BASED NEXT$SYM ADDRESS,
00462 1 HASH$MASK LIT '3FH',
00463 1 $TYPE LIT '2',
00464 1 DISPLACEMENT LIT '12',
00465 1 CCURS LIT '11',
00466 1 P$LENGTH LIT '3',
00467 1 FLD$LENGTH LIT '3',
00468 1 LEVEL LIT '10',
00469 1 REL$IC LIT '5',
00470 1 LOCATION LIT '2',
00471 1 START$NAME LIT '11', /*1 LESS*/
00472 1 FCB$ACCR LIT '4',
00473 1
00474 1
00475 1
00476 1 /* * * * * * SYMBOL TYPE LITERALS * * * * * */
00477 1
00478 1
00479 1 UNRESOLVED LIT '255',
00480 1 LABEL$TYPE LIT '32',
00481 1 MLLT$CCURS LIT '128',
00482 1 GROUP LIT '6',
00483 1 NON$NLMERICS$LIT LIT '7',
00484 1 ALPHA LIT '8',
00485 1 ALPHA$NUM LIT '9',
00486 1 LIT$SPACE LIT '10',
00487 1 LIT$CLCTE LIT '11',
00488 1 LIT$ZERO LIT '12',
00489 1 NUMERIC$LIT$FAL LIT '15',
00490 1 NUMERIC LIT '16',
00491 1 CCM$P LIT '21',
00492 1 AS$ED LIT '72',
00493 1 AS$SEC LIT '73',
00494 1 NUM$EC LIT '80',
00495 1
00496 1
00497 1 /* * * * * SYMBOL TABLE ROUTINES * * * */
00498 1
00499 1 SET$ADDRESS: PRCEDURE(ADDR);
00500 1 DECLARE ADDR ADDRESS;
00501 2 SYMBOL$ADDR(LOCATION)=ADDR;
00502 2 END SET$ADDRESS;
00503 1
00504 1 GET$ADDRESS: PRCEDURE ADDRESS;
00505 2 RETURN SYMBOL$ADDR(LOCATION);
00506 2 END GET$ADDRESS;
00507 1
00508 1 GET$FCB$ADDR: PRCEDURE ADDRESS;
00509 2 RETURN SYMBOL$ADDR(FCB$ADDR);
00510 2 END GET$FCB$ADDR;
00511 1
00512 1 GET$TYPE: PRCEDURE BYTE;
00513 2 RETURN SYMBOL($TYPE);
00514 2 END GET$TYPE;
00515 1
00516 1 SET$TYPE: PRCEDURE TYPE;
00517 2 DECLARE TYPE BYTE;
00518 2 SYMBOL($TYPE)=TYPE;
00519 2 END SET$TYPE;
00520 1
00521 1 GET$LENGTH: PRCEDURE ADDRESS;
00522 2 RETURN SYMBOL$ADDR(FLD$LENGTH);
00523 2 END GET$LENGTH;
00524 1
00525 1 GET$LEVEL: PRCEDURE BYTE;
00526 2 RETURN SHR(SYMBOL(LEVEL),4);
00527 2 END GET$LEVEL;
00528 1
00529 1 GET$DECIMAL: PRCEDURE BYTE;
00530 2 RETURN SYMBOL(LEVEL) AND OFH;
00531 2 END GET$DECIMAL;
00532 1
00533 1 GET$P$LENGTH: PRCEDURE BYTE;
00534 2 RETURN SYMBOL(P$LENGTH);
00535 2 END GET$P$LENGTH;
00536 1
00537 1 BUILD$SYMBOL: PRCEDURE(LEN);
00538 1 DECLARE LEN BYTE, TEMP ADDRESS;
00539 2 TEMP=NEXT$SYM;
00540 2 IF (NEXT$SYM=SYMBOL(LEN:=LEN + DISPLACEMENT))
00541 2 > MAX$MEMORY THEN CALL FATAL$ERROR('ST');
00542 2 CALL FILL(TEMP,0,LEN);
00543 2 END BUILD$SYMBOL;
00544 1

```



```

00001 1 AND:CLT:CCCLRS: PROCEDURE (TYPES:IN) BYTE;
00002 1 DECLARE TYPE:IN BYTE;
00003 1 RETURN TYPE:IN AND 127;
00004 1 END AND:CUT:CCCLRS;
00005 1
00006 1
00007 1
00008 1
00009 1
00010 1
00011 1
00012 1
00013 1
00014 1
00015 1
00016 1
00017 1
00018 1
00019 1
00020 1
00021 1
00022 1
00023 1
00024 1
00025 1
00026 1
00027 1
00028 1
00029 1
00030 1
00031 1
00032 1
00033 1
00034 1
00035 1
00036 1
00037 1
00038 1
00039 1
00040 1
00041 1
00042 1
00043 1
00044 1
00045 1
00046 1
00047 1
00048 1
00049 1
00050 1
00051 1
00052 1
00053 1
00054 1
00055 1
00056 1
00057 1
00058 1
00059 1
00060 1
00061 1
00062 1
00063 1
00064 1
00065 1
00066 1
00067 1
00068 1
00069 1
00070 1
00071 1
00072 1
00073 1
00074 1
00075 1
00076 1
00077 1
00078 1
00079 1
00080 1
00081 1
00082 1
00083 1
00084 1
00085 1
00086 1
00087 1
00088 1
00089 1
00090 1
00091 1
00092 1
00093 1
00094 1
00095 1
00096 1
00097 1
00098 1
00099 1
00100 1
00101 1
00102 1
00103 1
00104 1
00105 1
00106 1
00107 1
00108 1
00109 1
00110 1
00111 1
00112 1
00113 1
00114 1
00115 1
00116 1
00117 1
00118 1
00119 1
00120 1
00121 1
00122 1
00123 1
00124 1
00125 1
00126 1
00127 1
00128 1
00129 1
00130 1
00131 1
00132 1
00133 1
00134 1
00135 1
00136 1
00137 1
00138 1
00139 1
00140 1
00141 1
00142 1
00143 1
00144 1
00145 1
00146 1
00147 1
00148 1
00149 1
00150 1
00151 1
00152 1
00153 1
00154 1
00155 1
00156 1
00157 1
00158 1
00159 1
00160 1
00161 1
00162 1
00163 1
00164 1
00165 1
00166 1
00167 1
00168 1
00169 1
00170 1
00171 1
00172 1
00173 1
00174 1
00175 1
00176 1
00177 1
00178 1
00179 1
00180 1
00181 1
00182 1
00183 1
00184 1
00185 1
00186 1
00187 1
00188 1
00189 1
00190 1
00191 1
00192 1
00193 1
00194 1
00195 1
00196 1
00197 1
00198 1
00199 1
00200 1
00201 1
00202 1
00203 1
00204 1
00205 1
00206 1
00207 1
00208 1
00209 1
00210 1
00211 1
00212 1
00213 1
00214 1
00215 1
00216 1
00217 1
00218 1
00219 1
00220 1
00221 1
00222 1
00223 1
00224 1
00225 1
00226 1
00227 1
00228 1
00229 1
00230 1
00231 1
00232 1
00233 1
00234 1
00235 1
00236 1
00237 1
00238 1
00239 1
00240 1
00241 1
00242 1
00243 1
00244 1
00245 1
00246 1
00247 1
00248 1
00249 1
00250 1
00251 1
00252 1
00253 1
00254 1
00255 1
00256 1
00257 1
00258 1
00259 1
00260 1
00261 1
00262 1
00263 1
00264 1
00265 1
00266 1
00267 1
00268 1
00269 1
00270 1
00271 1
00272 1
00273 1
00274 1
00275 1
00276 1
00277 1
00278 1
00279 1
00280 1
00281 1
00282 1
00283 1
00284 1
00285 1
00286 1
00287 1
00288 1
00289 1
00290 1
00291 1
00292 1
00293 1
00294 1
00295 1
00296 1
00297 1
00298 1
00299 1
00300 1
00301 1
00302 1
00303 1
00304 1
00305 1
00306 1
00307 1
00308 1
00309 1
00310 1
00311 1
00312 1
00313 1
00314 1
00315 1
00316 1
00317 1
00318 1
00319 1
00320 1
00321 1
00322 1
00323 1
00324 1
00325 1
00326 1
00327 1
00328 1
00329 1
00330 1
00331 1
00332 1
00333 1
00334 1
00335 1
00336 1
00337 1
00338 1
00339 1
00340 1
00341 1
00342 1
00343 1
00344 1
00345 1
00346 1
00347 1
00348 1
00349 1
00350 1
00351 1
00352 1
00353 1
00354 1
00355 1
00356 1
00357 1
00358 1
00359 1
00360 1
00361 1
00362 1
00363 1
00364 1
00365 1
00366 1
00367 1
00368 1
00369 1
00370 1
00371 1
00372 1
00373 1
00374 1
00375 1
00376 1
00377 1
00378 1
00379 1
00380 1
00381 1
00382 1
00383 1
00384 1
00385 1
00386 1
00387 1
00388 1
00389 1
00390 1
00391 1
00392 1
00393 1
00394 1
00395 1
00396 1
00397 1
00398 1
00399 1
00400 1
00401 1
00402 1
00403 1
00404 1
00405 1
00406 1
00407 1
00408 1
00409 1
00410 1
00411 1
00412 1
00413 1
00414 1
00415 1
00416 1
00417 1
00418 1
00419 1
00420 1
00421 1
00422 1
00423 1
00424 1
00425 1
00426 1
00427 1
00428 1
00429 1
00430 1
00431 1
00432 1
00433 1
00434 1
00435 1
00436 1
00437 1
00438 1
00439 1
00440 1
00441 1
00442 1
00443 1
00444 1
00445 1
00446 1
00447 1
00448 1
00449 1
00450 1
00451 1
00452 1
00453 1
00454 1
00455 1
00456 1
00457 1
00458 1
00459 1
00460 1
00461 1
00462 1
00463 1
00464 1
00465 1
00466 1
00467 1
00468 1
00469 1
00470 1
00471 1
00472 1
00473 1
00474 1
00475 1
00476 1
00477 1
00478 1
00479 1
00480 1
00481 1
00482 1
00483 1
00484 1
00485 1
00486 1
00487 1
00488 1
00489 1
00490 1
00491 1
00492 1
00493 1
00494 1
00495 1
00496 1
00497 1
00498 1
00499 1
00500 1
00501 1
00502 1
00503 1
00504 1
00505 1
00506 1
00
```



```

00654 1      /* LENGTH 10 */
00655 1      RR LIT '54', /* READ RELATIVE SEQUENTIAL */
00656 1      WR LIT '55', /* WRITE RELATIVE SEQUENTIAL */
00657 1      RP LIT '56', /* READ RELATIVE RANDOM */
00658 1      WP LIT '57', /* WRITE RELATIVE RANDOM */
00659 1      RW LIT '58', /* REWRITE RELATIVE */
00660 1      DL LIT '59', /* DELETE RELATIVE */
00661 1
00662 1      /* LENGTH ELEVEN */
00663 1      MED LIT '60', /* MCVE EDITED */
00664 1
00665 1      /* LENGTH THIRTEEN */
00666 1      MNE LIT '61', /* MOVE NUMERIC EDITED */
00667 1
00668 1      /* VARIABLE LENGTH */
00669 1      GDF LIT '62', /* GO DEPENDING ON */
00670 1
00671 1      /* BUILD DIRECTING ONLY */
00672 1      INT LIT '63', /* INITIALIZE STORAGE */
00673 1      BST LIT '64', /* BACK STUFF ADDRESS */
00674 1      TER LIT '65', /* TERMINATE BUILD */
00675 1      SCD LIT '66', /* SET CODE START */
00676 1
00677 1      /* * * * * * PARSER ROUTINES * * * * */
00678 1
00679 1      DIGIT: PROCEDURE (CHAR) BYTE;
00680 1      DECLARE CHAR BYTE;
00681 1      RETURN (CHAR<='9') AND (CHAR>='0');
00682 1      END DIGIT;
00683 1
00684 1      LETTER: PROCEDURE BYTE;
00685 1      RETURN (CHAR>='A') AND (CHAR<='Z');
00686 1      END LETTER;
00687 1
00688 1      INVALID$TYPE: PROCEDURE;
00689 1      CALL PRINT$ERROR('IT');
00690 1      END INVALID$TYPE;
00691 1
00692 1      BYTE$CLT: PROCEDURE(CNE$BYTE);
00693 1      DECLARE CNE$BYTE BYTE;
00694 1      IF (OUTPUT$PTR=OUTPUT$PTR + 1) > OUTPUT$ENC THEN
00695 1      CC;
00696 1      CALL WRITE$OUTPUT(.OUTPUT$BUFF);
00697 1      OUTPUT$PTR=.OUTPUT$BUFF;
00698 1
00699 1      ENO;
00700 1      CLT$PTR$CHAR=CNE$BYTE;
00701 1      END BYTE$CLT;
00702 1
00703 1      ADDR$CLT: PROCEDURE (ADDR);
00704 1      DECLARE ADDR ADDRESS;
00705 1      CALL BYTE$CLT(LOW(ADDR));
00706 1      CALL BYTE$CLT(HIGH(ADDR));
00707 1      END ADDR$CLT;
00708 1
00709 1      INC$CLCNT: PROCEDURE(CNT);
00710 1      DECLARE CNT BYTE;
00711 1      IF (NEXT$AVAILABLE=NEXT$AVAILABLE + CNT)
00712 1      >MAX$INT$MEM THEN CALL FATAL$ERROR('MC');
00713 1      END INC$CLCNT;
00714 1
00715 1
00716 1      CNE$ADDR$OPP: PROCEDURE(CODE,ADDR);
00717 1      DECLARE CNE$CODE BYTE, ADDR ADDRESS;
00718 1      CALL BYTE$CLT(CODE);
00719 1      CALL ADDR$CLT(ADDR);
00720 1      CALL INC$CLCNT(3);
00721 1      END CNE$ADDR$OPP;
00722 1
00723 1      NCT$IMPLIMENTED: PROCEDURE;
00724 1      CALL PRINT$ERROR('NI');
00725 1      END NCT$IMPLIMENTED;
00726 1
00727 1
00728 1      MATCH: PROCEDURE ADDRESS;
00729 1      /* CHECKS AN IDENTIFIER TO SEE IF IT IS IN THE SYMBOL
00730 1      TABLE. IF IT IS PRESENT, CUR$SYM IS SET FOR ACCESS,
00731 1      OTHERWISE THE POINTERS ARE SET FOR ENTRY */
00732 1      DECLARE (PCINT,COLLISION$BASED PCINT) ADDRESS, (HOLD,I) BYTE;
00733 1      IF VARC>MAX$ID$LEN THEN
00734 1      VARC=MAX$ID$LEN;
00735 1
00736 1      HCLD=0;
00737 1      CC I=1 TO VARC;
00738 1      HOLD=HCLD+VARC(I);
00739 1
00740 1      PCINT=HASH$TAB$ADDR + SHL((HCLD AND HASH$MASK),1);
00741 1      DO FOREVER;
00742 1      IF CCLLISCN=0 THEN
00743 1      DO;
00744 1      CLP$SYM,COLLISION=NEXT$SYM;
00745 1      CALL BUILD$SYMBOL(VARC);
00746 1      SYMBCL(P$LENGTH)=VARC;
00747 1      DO I=1 TO VARC;
00748 1      SYMBCL(START$NAME+I)=VARC(I);
00749 1      END;
00750 1      CALL SET$TYPE(UNRESOLVED); /* UNRESOLVED LABEL */
00751 1      RETURN CUR$SYM;
00752 1      ELSE
00753 1      DO;
00754 1      CUR$SYM=CCLLISCN;
00755 1      IF (HOLD=GET$P$LENGTH)=VARC THEN
00756 1      CC;
00757 1      I=I+1;
00758 1      DO WHILE SYMBOL(START$NAME + I)= VARC(I);
00759 1      IF (I=I+1)>HCLD THEN RETURN(CUR$SYM=CCLLISCN);
00760 1      END;
00761 1      END;
00762 1      END;
00763 1      PCINT=CCLLISCN;
00764 1      END MATCH;
00765 1

```



```

00767 1 SET$VALUE: PROCEDURE(NUB);
00768 2 DECLARE NUB ADDRESS;
00769 2 VALUE(NUB)=NUB;
00770 2 END SET$VALUE;
00771 1
00772 1 SET$VALUE2: PROCEDURE(ADDR);
00773 2 DECLARE ADDR ADDRESS;
00774 2 VALUE2(NUB)=ADDR;
00775 2 END SET$VALUE2;
00776 1
00777 1
00778 1 SUB$CNT: PROCEDURE BYTE;
00779 2 IF (SUB$INC:=SUB$IND + 1)>8 THEN
00780 3 SUB$INC=1;
00781 2 RETURN SUB$IND;
00782 2 END SUB$CNT;
00783 1
00784 1
00785 1 CCDE$BYTE: PROCEDURE (CODE);
00786 2 DECLARE CODE BYTE;
00787 2 CALL BYTE$CLT(CODE);
00788 2 CALL INC$CLNT(1);
00789 2 END CCDE$BYTE;
00790 1
00791 1
00792 1 CODE$ADDRESS: PROCEDURE (CODE);
00793 2 DECLARE CODE ADDRESS;
00794 2 CALL ADDR$CLT(CODE);
00795 2 CALL INC$CLNT(2);
00796 2 END CODE$ADDRESS;
00797 1
00798 1
00799 1 INPL$NUMERIC: PROCEDURE BYTE;
00800 2 CC CTR=1 TO VAPC;
00801 3 IF NOT DIGIT(VARC(CTR)) THEN RETURN FALSE;
00802 3 END;
00803 2 RETURN TRUE;
00804 2 END INPL$NUMERIC;
00805 1
00806 1
00807 1 CCNVERT$INTEGER: PROCEDURE ADDRESS;
00808 2 ACTR=0;
00809 2 CC CTR=1 TO VARC;
00810 3 IF NOT DIGIT(VARC(CTR)) THEN CALL PRINT$ERROR('NA');
00811 3 ACTR=SHL(ACTR,3)+SHL(ACTR,1) + VARC(CTR) - '0';
00812 3 END;
00813 2 RETURN ACTR;
00814 2 END CCNVERT$INTEGER;
00815 1
00816 1
00817 1 BACK$STUFF: PROCEDURE (ADD1,ADD2);
00818 2 DECLARE (ADD1,ADD2) ADDRESS;
00819 2 CALL BYTE$CLT(BST);
00820 2 CALL ADDR$CLT(ADD1);
00821 2 CALL ADDR$CLT(ADD2);
00822 2 END BACK$STUFF;
00823 1
00824 1
00825 1 UNRESOLVED$BRANCH: PROCEDURE;
00826 2 CALL SET$VALUE(NEXT$AVAILABLE + 1);
00827 2 CALL CNE$ACR$OPP(BRN,0);
00828 2 CALL SET$VALUE2(NEXT$AVAILABLE);
00829 2 END UNRESOLVED$BRANCH;
00830 1
00831 1
00832 1 BACK$CCNC: PROCEDURE;
00833 2 CALL BACK$STUFF(VALUE(SP-1),NEXT$AVAILABLE);
00834 2 END BACK$CCNC;
00835 1
00836 1
00837 1 SET$BRANCH: PROCEDURE;
00838 2 CALL SET$VALUE(NEXT$AVAILABLE);
00839 2 CALL CODE$ADDRESS(0);
00840 2 END SET$BRANCH;
00841 1
00842 1
00843 1 KEEP$VALUES: PROCEDURE;
00844 2 CALL SET$VALUE(VALUE(SP));
00845 2 CALL SET$VALUE2(VALUE2(SP));
00846 2 END KEEP$VALUES;
00847 1
00848 1
00849 1 STANDARD$ATTRIBUTES: PROCEDURE(TYPE);
00850 2 DECLARE TYPE BYTE;
00851 2 CALL CODE$ADDRESS(GET$FCB$ADDR);
00852 2 CALL CODE$ADDRESS(GET$ADDRESS);
00853 2 CALL CODE$ADDRESS(GET$LENGTH);
00854 2 IF TYPE=0 THEN RETURN;
00855 2 CUR$SYM=SYMBOL$ACR(REL$ID);
00856 2 CALL CODE$ADDRESS(GET$ADDRESS);
00857 2 CALL CODE$EYTE(GET$LENGTH);
00858 2 END STANDARD$ATTRIBUTES;
00859 1
00860 1
00861 1 READ$WRITE: PROCEDURE(INDEX);
00862 2 DECLARE INDEX BYTE;
00863 2
00864 2 IF (CTR:=GET$TYPE)=1 THEN
00865 3 DC;
00866 3 CALL CODE$BYTE(ROF+INDEX);
00867 3 CALL STANDARD$ATTRIBUTES(0);
00868 3 END;
00869 2 ELSE IF CTR=2 THEN
00870 3 CC;
00871 3 CALL CODE$BYTE(ORS+INDEX);
00872 3 CALL STANDARD$ATTRIBUTES(1);
00873 3 END;
00874 2 ELSE IF CTR=3 THEN
00875 3 CC;
00876 3 CALL CODE$BYTE(QRR+INDEX);
00877 3 CALL STANDARD$ATTRIBUTES(1);
00878 3 END;

```



```

00879 2      ELSE IF CTR=4 THEN
00880 2      CC;
00881 2      CALL CCCE$BYTE(RVL+INDEX);
00882 3      CALL STANDARD$ATTRIBUTES(0);
00883 3      END;
00884 2      ELSE CALL PRINT$ERROR('FT');
00885 2      END READ$WRITE;
00886 1
00887 1
00888 1  ARITHMETIC$TYPE: PROCEDURE BYTE;
00889 1  IF ((L$TYPE=AND$CUT$OCCURS(L$TYPE))>=NUMERIC$LITERAL)
00890 2      OR (L$TYPE<=COMP) THEN RETURN L$TYPE - NUMERIC$LITERAL;
00891 2      CALL INVALID$TYPE;
00892 2      RETURN C;
00893 2      END ARITHMETIC$TYPE;
00894 1
00895 1
00896 1  DEL$RWT: PROCEDURE (FLAG);
00897 1  DECLARE FLAG BYTE;
00898 2      IF (CTR:=GET$TYPE)=3 THEN
00899 2      CC;
00900 3      IF FLAG THEN CALL CODE$BYTE(RWR);
00901 3      ELSE CALL CODE$BYTE(DLR);
00902 3      CALL STANDARD$ATTRIBUTES(1);
00903 3      RETURN;
00904 3      END;
00905 3      IF (CTR=2) AND (NOT FLAG) THEN CALL CODE$BYTE(DLS);
00906 2      ELSE IF (CTR>4) AND FLAG THEN CALL CODE$BYTE(RWS);
00907 2      ELSE CALL INVALID$TYPE;
00908 2      CALL STANDARD$ATTRIBUTES(0);
00909 2      END DEL$RWT;
00910 1
00911 1
00912 1  ATTRIBUTES: PROCEDURE;
00913 1  CALL CODE$ACCESS(L$ADDR);
00914 2      CALL CODE$BYTE(L$LENGTH);
00915 2      CALL CODE$BYTE(L$CEC);
00916 2      END ATTRIBUTES;
00917 1
00918 1
00919 1  LOAD$L$ID: PROCEDURE (S$PTR);
00920 2      DECLARE S$PTR BYTE;
00921 2      IF ((A$CTR:=VALUE(S$PTR))<NON$NUMERIC$LIT) OR
00922 2      (A$CTR=NUMERIC$LITERAL) THEN
00923 3      CC;
00924 3      L$ADDR=VALUE2(S$PTR);
00925 3      L$LENGTH=CC$LENGTH;
00926 3      L$TYPE=A$CTR;
00927 3      RETURN;
00928 3      END;
00929 3      IF A$CTR<=LIT$ZERO THEN
00930 2      CC;
00931 3      L$TYPE,L$ADDR=A$CTR;
00932 3      L$LENGTH=1;
00933 3      RETURN;
00934 3      END;
00935 3      CLR$SYM=VALUE(S$PTR);
00936 3      L$TYPE=GET$TYPE;
00937 3      L$LENGTH=GET$LENGTH;
00938 3      L$CEC=GET$CECIMAL;
00939 2      IF (L$ADDR:=VALUE2(S$PTR))=0 THEN L$ADDR=GET$ADDRESS;
00940 2      END LOAD$L$ID;
00941 1
00942 1
00943 1
00944 1  LOAD$REG: PROCEDURE (REG$NO,PTR);
00945 1  DECLARE (REG$NO,PTR) BYTE;
00946 2      CALL LOAD$L$ID(PTR);
00947 2      CALL CODE$BYTE(LCD+ARITHMETIC$TYPE);
00948 2      CALL ATTRIBUTES;
00949 2      CALL CODE$BYTE(REG$NO);
00950 2      END LOAD$REG;
00951 1
00952 1
00953 1  STORE$REG: PROCEDURE (PTR);
00954 1  DECLARE PTR BYTE;
00955 2      CALL LOAD$L$ID(PTR);
00956 2      CALL CODE$BYTE(STO + ARITHMETIC$TYPE -1);
00957 2      CALL ATTRIBUTES;
00958 2      END STORE$REG;
00959 1
00960 1
00961 1  STORE$CONSTANT: PROCEDURE ADDRESS;
00962 2      IF (MAX$INT$MEM:=MAX$INT$MEM - VARC)<NEXT$AVAILABLE
00963 2      THEN CALL FATAL$ERROR('MD');
00964 2      CALL BYTE$CLT(INT);
00965 2      CALL ADDR$CLT(MAX$INT$MEM);
00966 2      CALL ADDR$CLT(CCN$LENGTH:=VARC);
00967 2      DO CTR = 1 TO CCN$LENGTH;
00968 3      CALL BYTE$CLT(VARC(CTR));
00969 3      END;
00970 2      RETURN MAX$INT$MEM;
00971 2      END STORE$CONSTANT;
00972 1
00973 1
00974 1  NUMERIC$LIT: PROCEDURE BYTE;
00975 1  DECLARE CHAR BYTE;
00976 2      CC CTR=1 TO VARC;
00977 3      IF NOT (DIGIT(CHAR:=VARC(CTR))
00978 3      OR (CHAR='-') OR (CHAR='+')
00979 3      OR (CHAR='.')) THEN RETURN FALSE;
00980 3      END;
00981 3      RETURN TRUE;
00982 2      END NUMERIC$LIT;
00983 1
00984 1
00985 1  FUNC$STORE: PROCEDURE;
00986 2      IF VALUE(SP)<>0 THEN
00987 2      CC;
00988 3      CALL CCCE$BYTE(RND);
00989 3      CALL CCCE$BYTE(L$DEC);
00990 3      END;
00991 2      CALL STORE$REG(SP-1);
00992 2      END FUNC$STORE;

```



```

00594 1
00595 1
00596 1
00597 1
00598 1
00599 1
01000 1
01001 1
01002 1
01003 1
01004 1
01005 1
01006 1
01007 1
01008 1
01009 1
01010 1
01011 1
01012 1
01013 1
01014 1
01015 1
01016 1
01017 1
01018 1
01019 1
01020 1
01021 1
01022 1
01023 1
01024 1
01025 1
01026 1
01027 1
01028 1
01029 1
01030 1
01031 1
01032 1
01033 1
01034 1
01035 1
01036 1
01037 1
01038 1
01039 1
01040 1
01041 1
01042 1
01043 1
01044 1
01045 1
01046 1
01047 1
01048 1
01049 1
01050 1
01051 1
01052 1
01053 1
01054 1
01055 1
01056 1
01057 1
01058 1
01059 1
01060 1
01061 1
01062 1
01063 1
01064 1
01065 1
01066 1
01067 1
01068 1
01069 1
01070 1
01071 1
01072 1
01073 1
01074 1
01075 1
01076 1
01077 1
01078 1
01079 1
01080 1
01081 1
01082 1
01083 1
01084 1
01085 1
01086 1
01087 1
01088 1
01089 1
01090 1
01091 1
01092 1
01093 1
01094 1
01095 1
01096 1
01097 1
01098 1
01099 1

ADCSUB: PROCEDURE (INDEX);
  DECLARE INDEX BYTE;
  CALL LCAC$FEC(0,MPP1);
  IF VALUE(SP-3)<>0 THEN
    CC;
    CALL LCAD$REG(1,SP-3);
    CALL CCCE$BYTE(ADD);
    CALL CCCE$BYTE(STI);
  END;
  CALL LCAC$FEC(1,SP-1);
  CALL CCCE$BYTE(ADD + INDEX);
  CALL RCLND$STORE;
ENC ADCSUB;

MULT$CIV: PROCEDURE (INDEX);
  DECLARE INDEX BYTE;
  CALL LCAC$FEC(0,MPP1);
  CALL LCAC$FEC(1,SP-1);
  CALL CCCE$BYTE(MUL + INDEX);
  CALL RCLND$STORE;
ENC MLLT$CIV;

CHECK$SUBSCRIPT: PROCEDURE;
  CLR$SYM=VALUE(MP);
  IF GET$TYPE<MLL$CCURS THEN
    CC;
    CALL PRINT$ERRGR('IS');
    RETURN;
  END;
  IF !NPLT$NUMERIC THEN
    CC;
    CALL SET$VALUE2(GET$ADDRESS + (GET$LENGTH * CCNVERT$INTEGER));
    RETURN;
  END;
  CLR$SYM=MATCH;
  IF ((CTR:=GET$TYPE)<NUMERIC) OR (CTR>COMP) THEN
    CALL PRINT$ERRGR('TE');
    CALL CNE$ADCP$OPP(SCR,GET$ADDRESS);
    CALL CCCE$BYTE(SUB$CNT);
    CALL CCCE$BYTE(GET$LENGTH);
    CALL SET$VALUE2(SUB$IND);
  END CHECK$SUBSCRIPT;

LOAD$LABEL: PROCEDURE;
  CLR$SYM=VALUE(MP);
  IF (A$CTR:=GET$ADDRESS)<>0 THEN
    CALL BACK$STUFF(A$CTR,VALUE2(MP));
    CALL SET$ADDRESS(VALUE2(MP));
    CALL SET$TYPE(LABEL$TYPE);
    IF (A$CTR:=GET$FCB$ADDR)<>0 THEN
      CALL BACK$STUFF(A$CTR,NEXT$AVAILABLE);
      SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE;
      CALL CNE$ADCP$OPP(RET,0);
    END LCAC$LABEL;

LOAD$SEC$LABEL: PROCEDURE;
  A$CTR=VALUE(MP);
  CALL SET$VALUE(HOLD$SECTION);
  FOLD$SECTION=A$CTR;
  A$CTR=VALUE2(MP);
  CALL SET$VALUE2(HOLD$SEC$ADDR);
  FOLD$SEC$ADDR = A$CTR;
  CALL LCAC$LABEL;
ENC LOAD$SEC$LABEL;

LABEL$ADDR: PROCEDURE (ADDR,HOLD)ADDRESS;
  DECLARE ADDR ADDRESS;
  DECLARE FOLD BYTE;
  CLR$SYM=ADDR;
  IF (CTR:=GET$TYPE)=LABEL$TYPE THEN
    CC;
    IF FOLD THEN RETURN GET$ADDRESS;
    RETURN GET$FCB$ADDR;
  END;
  IF CTR<>LNRESOLVED THEN CALL INVALID$TYPE;
  IF FOLD THEN
    CC;
    A$CTR=GET$ADDRESS;
    CALL SET$ADDRESS(NEXT$AVAILABLE + 1);
    RETURN A$CTR;
  END;
  END;
  A$CTR=GET$FCB$ADDR;
  SYMBOL$ADDR(FCB$ADDR)=NEXT$AVAILABLE + 1;
  RETURN A$CTR;
END LABEL$ADDR;

CODE$FCP$DISPLAY: PROCEDURE (PCINT);
  DECLARE PCINT BYTE;
  CALL LCAC$FEC(PCINT);
  CALL CNE$ADCP$OPP(DIS,L$ADDR);
  CALL CCCE$BYTE(L$LENGTH);
ENC CODE$FCP$DISPLAY;

A$AN$TYPE: PROCEDURE BYTE;
  RETURN (L$TYPE=ALPHA) OR (L$TYPE=ALPHASNUM);
ENC A$AN$TYPE;

```



```

01100 1 NOT$INTEGER: PROCEDURE BYTE;
01101 2 RETURN L$DEC<>0;
01102 2 END ACT$INTEGER;
01103 1
01104 1
01105 1 NUMERIC$TYPE: PROCEDURE BYTE;
01106 2 RETURN (L$TYPE>=NUMERIC) AND (L$TYPE<=COMP);
01107 2 END NUMERIC$TYPE;
01108 1
01109 1
01110 1 GEN$COMPARE: PROCEDURE;
01111 2 DECLARE (H$TYPE,H$DEC) BYTE;
01112 2 (H$ADDR,H$LENGTH) ADDRESS;
01113 2
01114 2 CALL LCAD$LCID(MP);
01115 2 L$TYPE=AND$CLT$OCCURS(L$TYPE);
01116 2 IF CCND$TYPE=3 THEN /* COMPARE FOR NUMERIC */
01117 2 DO;
01118 3 IF A$AN$TYPE OR (L$TYPE>COMP) THEN CALL INVALID$TYPE;
01119 3 IF L$TYPE=NUMERIC THEN CALL CODE$BYTE(CNU);
01120 3 ELSE CALL CODE$BYTE(CNS);
01121 3 CALL CODE$ADDRESS(L$ADDR);
01122 3 CALL CODE$ADDRESS(L$LENGTH);
01123 3 CALL SET$BRANCH;
01124 2 END;
01125 2 ELSE IF CCND$TYPE=4 THEN
01126 2 DO;
01127 3 IF NUMERIC$TYPE THEN CALL INVALID$TYPE;
01128 3 CALL CODE$BYTE(CAL);
01129 3 CALL CODE$ADDRESS(L$ADDR);
01130 3 CALL CODE$ADDRESS(L$LENGTH);
01131 3 CALL SET$BRANCH;
01132 2 END;
01133 2 ELSE
01134 3 DO;
01135 4 IF NUMERIC$TYPE THEN CTR=1;
01136 4 ELSE CTR=0;
01137 4 H$TYPE=L$TYPE;
01138 4 H$DEC=L$DEC;
01139 4 H$ADDR=L$ADDR;
01140 4 H$LENGTH=L$LENGTH;
01141 4 CALL LCAD$LCID(SP);
01142 4 IF NUMERIC$TYPE THEN CTR=CTR+1;
01143 4 IF CTR=2 THEN /* NUMERIC COMPARE */
01144 5 DO;
01145 6 CALL LGAD$REG(O,MP);
01146 6 CALL LGAD$REG(I,SP);
01147 6 CALL CODE$BYTE(SUB);
01148 6 CALL CODE$BYTE(RGT + CCND$TYPE);
01149 6 CALL SET$BRANCH;
01150 5 END;
01151 4 ELSE DO;
01152 5 /* ALPHA NUMERIC COMPARE */
01153 6 IF (H$DEC<>0) OR (H$TYPE=COMP)
01154 7 OR (L$DEC<>0) OR (L$TYPE=COMP)
01155 7 OR (H$LENGTH<>L$LENGTH) THEN CALL INVALID$TYPE;
01156 6 CALL CODE$BYTE(SGT+CCND$TYPE);
01157 6 CALL CODE$ADDRESS(H$ADDR);
01158 6 CALL CODE$ADDRESS(L$ADDR);
01159 6 CALL CODE$ADDRESS(H$LENGTH);
01160 5 END;
01161 4 END;
01162 2 END GEN$COMPARE;
01163 1
01164 1 MOVE$TYPE: PROCEDURE BYTE;
01165 2 DECLARE
01166 2 H$LD$TYPE BYTE;
01167 2 ALPHASNUM$MOVE LIT '0';
01168 2 AS$N$ED$MOVE LIT '1';
01169 2 NUMERIC$MOVE LIT '2';
01170 2 N$ED$MOVE LIT '3';
01171 2
01172 2 L$TYPE=AND$CLT$OCCURS(L$TYPE);
01173 2 IF ((HOLD$TYPE=AND$CLT$OCCURS(GET$TYPE))=GROUP) OR (L$TYPE=GROUP)
01174 2 THEN RETURN ALPHASNUM$MOVE;
01175 2 IF HOLD$TYPE=ALPHA THEN
01176 3 IF A$AN$TYPE OR (L$TYPE=AS$ED) OR (L$TYPE=AS$N$ED)
01177 3 THEN RETURN ALPHASNUM$MOVE;
01178 2 IF HOLD$TYPE=ALPHASNUM THEN
01179 3 DO;
01180 4 IF NOT$INTEGER THEN CALL INVALID$TYPE;
01181 4 RETURN ALPHASNUM$MOVE;
01182 3 END;
01183 2 IF (HOLD$TYPE>=NUMERIC) AND (HOLD$TYPE<=COMP) THEN
01184 3 DO;
01185 4 IF (L$TYPE=ALPHA) OR (L$TYPE>COMP) THEN CALL INVALID$TYPE;
01186 4 RETURN NUMERIC$MOVE;
01187 3 END;
01188 2 IF HOLD$TYPE=AS$N$ED THEN
01189 3 DO;
01190 4 IF NOT$INTEGER THEN CALL INVALID$TYPE;
01191 4 RETURN AS$N$ED$MOVE;
01192 3 END;
01193 2 IF HOLD$TYPE=AS$ED THEN
01194 3 IF A$AN$TYPE OR (L$TYPE>COMP) THEN RETURN AS$N$ED$MOVE;
01195 3 IF HOLD$TYPE=NUM$ED THEN
01196 4 IF NUMERIC$TYPE OR (L$TYPE=ALPHASNUM) THEN
01197 5 RETURN N$ED$MOVE;
01198 4 CALL INVALID$TYPE;
01199 3 RETURN C;
01200 2 END MOVE$TYPE;
01201 1

```



```

01203 1 GEN$MCVE:PROCEDURE;
01204 DECLARE
01205 LENGTH1 ADDRESS,
01206 ADDR1 ADDRESS,
01207 EXTRA ADDRESS;
01208
01209 ACC$ACD$LEN: PROCEDURE;
01210 CALL CCCE$ADDRESS(ADDR1);
01211 CALL CCCE$ADDRESS(L$ADDR1);
01212 CALL CCCE$ADDRESS(L$LENGTH1);
01213 ENC ADC$ACC$LEN;
01214
01215 CCCE$FCR$ECIT: PROCEDURE;
01216 CALL ACC$ADD$LEN;
01217 CALL CCCE$ADDRESS(GET$FCR$ADDR);
01218 CALL CCCE$ADDRESS(LENGTH1);
01219 ENC CCCE$FCR$ECIT;
01220
01221 CALL LOAD$LC$(MPPI);
01222 CLR$SYM=VALUE(SP);
01223 IF (ADDR1=VALUE2(SP))=0 THEN ADDR1=GET$ADDRESS;
01224 LENGTH1=GET$LENGTH;
01225
01226 CC CASE MCVE$TYPE;
01227
01228 /* ALPHA NUMERIC MOVE */
01229 DO;
01230 IF LENGTH1>L$LENGTH THEN EXTRA=LENGTH1-L$LENGTH;
01231 ELSE DO;
01232 EXTRA=0;
01233 L$LENGTH=LENGTH1;
01234 END;
01235 CALL CODE$BYTE(MOV);
01236 CALL ADD$ADD$LEN;
01237 CALL CODE$ADDRESS(EXTRA);
01238 ENC;
01239
01240 /* ALPHA NUMERIC EDITED */
01241 DO;
01242 CALL CODE$BYTE(MED);
01243 CALL CODE$FOR$EDIT;
01244 ENC;
01245
01246 /* NUMERIC MCVE */
01247 DO;
01248 CALL LOAD$REG(2,MPPI);
01249 CALL STORE$REG(SP);
01250 ENC;
01251
01252 /* NUMERIC EDITED MOVE */
01253 DO;
01254 CALL CCCE$BYTE(MNE);
01255 CALL CCCE$FOR$EDIT;
01256 CALL CCCE$BYTE(L$SEC);
01257 CALL CCCE$BYTE(GET$DECIMAL);
01258 ENC;
01259 END GEN$MOVE;
01260
01261 CODE$GEN: PROCEDURE(PRODUCTION);
01262 DECLARE PROCLCTIGN BYTE;
01263 IF PRINT$PRCC THEN
01264 CC;
01265 CALL CRLF;
01266 CALL PRINT$CHAR(PCUND);
01267 CALL PRINT$NUMBER(PRODUCTION);
01268 ENC;
01269
01270 CC CASE PROCLCTIGN;
01271
01272 /* P R O D U C T I C N S */
01273 /* CASE G A C T I O N S */
01274 ;
01275
01276 /* 1 <P-DIV> ::= PROCEDURE DIVISION <USING> . <PRCC-BODY> */
01277 CC;
01278 COMPILING = FALSE;
01279 IF SECTION$FLAG THEN CALL LOAD$SEC$LABEL;
01280 ENC;
01281
01282 /* 2 <USING> ::= USING <ID-STRING> */
01283 CALL NCT$IMPLIMENTED; /* INTER PROG COMM */
01284
01285 /* 3 <EMPTY> */
01286 ; /* NO ACTION REQUIRED */
01287
01288 /* 4 <IC-STRING> ::= <IO> */
01289
01290
01291
01292
01293
01294
01295
01296
01297
01298
01299
01300

```



```

013002  /* 5 <ID-STRING> <ID> */
013003
013004 CC;
013005     IF (ID$PTR:=ICPTR+1)=20 THEN
013006     DO;
013007         CALL PRINT$ERROR('ID');
013008         IC$PTR=19;
013009     END;
013010     ID$STACK(ID$PTR)=VALUE(SP);
013011 END;
013012
013013 /* 6 <PRCC-BODY> ::= <PARAGRAPH> */
013014 ;
013015 /* NO ACTION REQUIRED */
013016
013017 /* 7 <PRCC-BODY> <PARAGRAPH> */
013018 ;
013019 /* NO ACTION REQUIRED */
013020
013021 /* 8 <PARAGRAPH> ::= <IC> . <SENTENCE-LIST> */
013022
013023 CC;
013024     IF SECTION$FLAG=0 THEN SECTION$FLAG=2;
013025     CALL LCAC$LABEL;
013026 END;
013027
013028 /* 9 <ID> SECTION . */
013029
013030 CC;
013031     IF SECTION$FLAG<>1 THEN
013032     DO;
013033         IF SECTION$FLAG=2 THEN CALL PRINT$ERROR('PF');
013034         SECTION$FLAG=1;
013035         FCLC$SECTION=VALUE(MP);
013036         FCLC$SEC$ADDR=VALUE2(MP);
013037     END;
013038     ELSE CALL LCAD$SEC$LABEL;
013039 END;
013040
013041 /* 10 <SENTENCE-LIST> ::= <SENTENCE> . */
013042 ;
013043 /* NO ACTION REQUIRED */
013044
013045 /* 11 <SENTENCE-LIST> <SENTENCE> . */
013046 ;
013047 /* NO ACTION REQUIRED */
013048
013049 /* 12 <SENTENCE> ::= <IMPERATIVE> */
013050 ;
013051 /* NO ACTION REQUIRED */
013052
013053 /* 13 <CONDITIONAL> */
013054 ;
013055 /* NO ACTION REQUIRED */
013056
013057 /* 14 ENTER <ID> <OPT-IC> */
013058
013059 CALL NOT$IMPLIMENTED; /* LANGUAGE CHANGE */
013060
013061 /* 15 <IMPERATIVE> ::= ACCEPT <SUBID> */
013062
013063 CC;
013064     CALL LCAC$L$IC(SP);
013065     CALL CNE$ADDR$CPP(ACC,L$ADDR);
013066     CALL CCCE$BYTE(L$LENGTH);
013067 END;
013068
013069 /* 16 <ARITHMETIC> */
013070 ;
013071 /* NO ACTION REQUIRED */
013072
013073 /* 17 CALL <LIT> <USING> */
013074
013075 CALL NOT$IMPLIMENTED; /* INTER PROG COMM */
013076
013077 /* 18 CLOSE <ID> */
013078
013079 CALL CNE$ADDR$OPP(CLS,GET$FCB$ADDR);
013080
013081 /* 19 <FILE-ACT> */
013082 ;
013083 /* NO ACTION REQUIRED */
013084
013085 /* 20 DISPLAY <LIT/ID> <OPT-LIT/IC> */
013086
013087 CC;
013088     CALL CODE$FOR$DISPLAY(MPP1);
013089     IF VALUE(SP)<>0 THEN CALL CODE$FOR$DISPLAY(SP);
013090 END;
013091
013092 /* 21 EXIT <PROGRAM-ID> */
013093 ;
013094 /* NO ACTION REQUIRED */
013095
013096 /* 22 GO <ID> */
013097
013098 CALL CNE$ADDR$OPP(BRN,LABEL$ADDR(VALUE(SP),1));
013099

```



```

C1400 3 /*      23      GC <ID-STRING> DEPENDING <IC>      */
C1401 3
C1402 3 CC;
C1403 3     CALL CCCE$BYTE(GDP);
C1404 4     CALL CCCE$BYTE(ID$PTR);
C1405 4     CUR$SYN=VALUE(SP);
C1406 4     CALL CCCE$BYTE(GET$LENGTH);
C1407 4     CALL CCCE$ADDRESS(GET$ADDRESS);
C1408 4     DC CTF=C TO ID$PTR;
C1409 4     CALL CCCE$ADDRESS(LABEL$ADDR(ID$STACK(ID$PTR),1));
C1410 5     ENC;
C1411 4     END;
C1412 3
C1413 3 /*      24      MOVE <LIT/ID> TO <SUBID>      */
C1414 3
C1415 3     CALL GEN$MCVE;
C1416 3
C1417 3 /*      25      OPEN <TYPE-ACTION> <ID>      */
C1418 3
C1419 3     CALL ONE$ACCR$CPP(CPN + VALUE(MPP1), GET$FCB$ACCR);
C1420 3
C1421 3 /*      26      PERFORM <ID> <THRU> <FINISH>      */
C1422 3
C1423 3 CC;
C1424 4     DECLARE (ACCR2,ADDR3) ADDRESS;
C1425 4     IF VALUE(SP-1)=0 THEN ADDR2=LABEL$ADDR(VALUE(MPP1),0);
C1426 4     ELSE ACCR2=LABEL$ADDR(VALUE(SP-1),0);
C1427 4     IF (ACCR3=VALUE2(SP))=0 THEN ADDR3=NEXT$AVAILABLE + 7;
C1428 4     ELSE CALL BACKSTUFF(VALUE(SP),NEXT$AVAILABLE + 7);
C1429 4     CALL CNE$ADDR$CPP(PCR,LABEL$ADDR(VALUE(MPP1),1));
C1430 4     CALL CCCE$ADDRESS(ACCR2);
C1431 4     CALL CCCE$ADDRESS(ADDR3);
C1432 4     ENC;
C1433 3
C1434 3 /*      27      <READ-IC>      */
C1435 3
C1436 3     CALL NCT$IMPLIMENTED; /* GRAMMAR ERROR */
C1437 3
C1438 3 /*      28      STOP <TERMINATE>      */
C1439 3
C1440 3 CC;
C1441 4     IF VALLE(SP)=0 THEN CALL CCCE$BYTE(STP);
C1442 4     ELSE CALL CNE$ACCR$DPP(STD,VALUE(SP));
C1443 4     ENC;
C1444 3
C1445 3 /*      29      <CCNDITIONAL> ::= <ARITHMETIC> <SIZE-ERROR>      */
C1446 3 /*      29      <IMPERATIVE>      */
C1447 3
C1448 3     CALL BACK$CCND;
C1449 3
C1450 3 /*      30      <FILE-ACT> <INVALID> <IMPERATIVE>      */
C1451 3
C1452 3     CALL BACK$CCND;
C1453 3
C1454 3 /*      31      IF <CONDITION> <ACTION> ELSE      */
C1455 3 /*      31      <IMPERATIVE>      */
C1456 3
C1457 3 CC;
C1458 4     CALL BACKSTUFF(VALUE(MPP1),VALUE2(SP-2));
C1459 4     CALL BACKSTUFF(VALUE(SP-2),NEXT$AVAILABLE);
C1460 4     ENC;
C1461 3
C1462 3 /*      32      <READ-ID> <SPECIAL> <IMPERATIVE>      */
C1463 3
C1464 3     CALL BACK$CCND;
C1465 3
C1466 3 /*      33      <ARITHMETIC> ::= ADD <L/ID> <OPT-L/ID> TO <SUBID>      */
C1467 3 /*      33      <RCUND>      */
C1468 3
C1469 3     CALL ADD$SUB(0);
C1470 3
C1471 3 /*      34      DIVIDE <L/IC> INTO <SUBID> <RCUND>      */
C1472 3
C1473 3     CALL MULT$CIV(1);
C1474 3
C1475 3 /*      35      MULTIPLY <L/ID> BY <SUBID> <RCUND>      */
C1476 3
C1477 3     CALL MULT$DIV(0);
C1478 3
C1479 3 /*      36      SUBTRACT <L/ID> <CPT-L/IC> FROM      */
C1480 3 /*      36      <SUBID> <ROUND>      */
C1481 3
C1482 3     CALL ADD$SLE(1);
C1483 3
C1484 3 /*      37      <FILE-ACT> ::= DELETE <ID>      */
C1485 3
C1486 3     CALL DEL$RWT(0);
C1487 3
C1488 3 /*      38      REWRITE <ID>      */
C1489 3
C1490 3     CALL DEL$RWT(1);
C1491 3
C1492 3 /*      39      WRITE <ID> <SPECIAL-ACT>      */
C1493 3
C1494 3     CALL REAC$WRITE(1);
C1495 3
C1496 3 /*      40      <CCNDITION> ::= <LIT/ID> <NOT> <CCND-TYPE>      */
C1497 3
C1498 3     CALL GEN$CCMPARE;
C1499 3
C1500 3 /*      41      <CCND-TYPE> ::= NUMERIC      */
C1501 3
C1502 3     CCND$TYPE=3;
C1503 3
C1504 3 /*      42      ALPHABETIC      */
C1505 3
C1506 3     CCND$TYPE=4;
C1507 3
C1508 3 /*      43      <CCMPARE> <LIT/ID>      */
C1509 3
C1510 3     CALL KEEP$VALUES;

```



```

C1511 3
C1512
C1513 /* 44 <NCT> ::= NOT */
C1514 CALL COCE$BYTE(NEG);
C1515
C1516 /* 45 <EMPTY> */
C1517 ; /* NO ACTION REQUIRED */
C1518
C1519 /* 46 <COMPARE> ::= GREATER */
C1520 CCND$TYPE=0;
C1521
C1522 /* 47 LESS */
C1523 CCND$TYPE=1;
C1524
C1525 /* 48 EQUAL */
C1526 CCND$TYPE=2;
C1527
C1528 /* 49 <PCUND> ::= ROUNDED */
C1529 CALL SET$VALUE(1);
C1530
C1531 /* 50 <EMPTY> */
C1532 ; /* NO ACTION REQUIRED */
C1533
C1534 /* 51 <TERMINATE> ::= <LITERAL> */
C1535 ; /* NO ACTION REQUIRED */
C1536
C1537 /* 52 RUN */
C1538 ; /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
C1539
C1540 /* 53 <SPECIAL> ::= <INVALID> */
C1541 ; /* NO ACTION REQUIRED */
C1542
C1543 /* 54 END */
C1544 CC;
C1545 CALL SET$VALUE(2);
C1546 CALL COCE$BYTE(EOR);
C1547 CALL SET$BRANCH;
C1548 END;
C1549
C1550 /* 55 <CPT-ID> ::= <SUBID> */
C1551 ; /* VALLE AND VALUE2 ALREADY SET */
C1552
C1553 /* 56 */
C1554 ; /* VALLE ALREADY ZERO */
C1555
C1556 /* 57 <ACTION> ::= <IMPERATIVE> */
C1557 CALL UNRESOLVED$BRANCH;
C1558
C1559 /* 58 NEXT SENTENCE */
C1560 CALL UNRESOLVED$BRANCH;
C1561
C1562 /* 59 <THRU> ::= THRU <ID> */
C1563 CALL KEEP$VALUES;
C1564
C1565 /* 60 */
C1566 ; /* NO ACTION REQUIRED */
C1567
C1568 /* 61 <FINISH> ::= <L/ID> TIMES */
C1569 CC;
C1570 CALL LCAC$LSID(MP);
C1571 CALL CNE$ADDR$COP(LDI,L$ADDR);
C1572 CALL COCE$BYTE(L$LENGTH);
C1573 CALL SET$VALUE2(NEXT$AVAILABLE);
C1574 CALL CNE$ADDR$COP(DEC,0);
C1575 CALL COCE$ADDRESS(0);
C1576 CALL SET$VALUE(NEXT$AVAILABLE);
C1577 END;
C1578
C1579 /* 62 UNTIL <CONDITION> */
C1580 CALL KEEP$VALUES;
C1581
C1582 /* 63 */
C1583 ; /* NO ACTION REQUIRED */
C1584
C1585 /* 64 <INVALID> ::= INVALID */
C1586 CC;
C1587 CALL SET$VALUE(1);
C1588 CALL COCE$BYTE(INV);
C1589 CALL SET$BRANCH;
C1590 END;
C1591
C1592 /* 65 <SIZE-ERROR> ::= SIZE ERROR */
C1593 CC;
C1594 CALL COCE$BYTE(SFR);
C1595 CALL UNRESOLVED$BRANCH;
C1596 END;
C1597
C1598
C1599
C1600
C1601
C1602
C1603
C1604
C1605
C1606
C1607
C1608
C1609
C1610
C1611
C1612
C1613
C1614
C1615
C1616
C1617
C1618

```



```

01619  /*      66  <SPECIAL-ACT> ::= <WHEN> ADVANCING <HOW-MANY>      */
01620  CALL NOT$IMPLIMENTED; /* CARRAGE CCNTROL */
01621
01622  /*      67      */
01623  ; /* NO ACTION REQUIRED */
01624
01625  /*      68  <WHEN> ::= BEFORE      */
01626  CALL NOT$IMPLIMENTED; /* CARRAGE CCNTROL */
01627
01628  /*      69      AFTER      */
01629  CALL NOT$IMPLIMENTED; /* CARRAGE CCNTROL */
01630
01631  /*      70  <HOW-MANY> ::= <INTEGER>      */
01632  CALL NOT$IMPLIMENTED; /* CARRAGE CCNTROL */
01633
01634  /*      71      PAGE      */
01635  CALL NOT$IMPLIMENTED; /* CARRAGE CCNTROL */
01636
01637  /*      72  <TYPE-ACTION> ::= INPUT      */
01638  ; /* NO ACTION REQUIRED - VALUE(SP) ALREADY ZERO */
01639
01640  /*      73      OUTPUT      */
01641  CALL SET$VALUE(1);
01642
01643  /*      74      I-C      */
01644  CALL SET$VALUE(2);
01645
01646  /*      75  <SUBID> ::= <SUBSCRIPT>      */
01647  ; /* VALLE AND VALUE2 ALREADY SET */
01648
01649  /*      76  <ID>      */
01650  ; /* NO ACTION REQUIRED */
01651
01652  /*      77  <INTEGER> ::= <INPUT>      */
01653  CALL SET$VALUE(CONVERT$INTEGER);
01654
01655  /*      78  <ID> ::= <INPUT>      */
01656  CC;
01657  IF SET$TYPE=UNRESOLVED THEN CALL SET$VALUE2(NEXT$AVAILABLE);
01658  END;
01659
01660  /*      79  <L/ID> ::= <INPUT>      */
01661  CC;
01662  IF NUMERIC$LIT THEN
01663  CC;
01664  CALL SET$VALUE(NUMERIC$LITERAL);
01665  CALL SET$VALUE2(STORE$CONSTANT);
01666  END;
01667  ELSE CALL SET$VALUE(MATCH);
01668  END;
01669
01670  /*      80      <SUBSCRIPT>      */
01671  ; /* NO ACTION REQUIRED */
01672
01673  /*      81      ZERO      */
01674  CALL SET$VALUE(LIT$ZERO);
01675
01676  /*      82  <SLBSCRIPT> ::= <ID> ( <INPUT> )      */
01677  CALL CHECK$SUBSCRIPT;
01678
01679  /*      83  <OPT-L/ID> ::= <L/ID>      */
01680  ; /* NO ACTION REQUIRED */
01681
01682  /*      84      <EMPTY>      */
01683  ; /* VALLE ALREADY SET */
01684
01685  /*      85  <NA-LIT> ::= <LIT>      */
01686  CC;
01687  CALL SET$VALUE(NON$NUMERIC$LIT);
01688  CALL SET$VALUE2(STORE$CCNSTANT);
01689  END;
01690
01691  /*      86      SPACE      */
01692  CALL SET$VALUE(LIT$SPACE);
01693
01694  /*      87      QUOTE      */
01695  CALL SET$VALUE(LIT$QUOTE);
01696
01697  /*      88  <LITERAL> ::= <NN-LIT>      */
01698  ; /* NO ACTION REQUIRED */
01699
01700
01701
01702
01703
01704
01705
01706
01707
01708
01709
01710
01711
01712
01713
01714
01715
01716
01717
01718
01719
01720
01721
01722
01723

```



```

C1724 3 /*      89          <INPUT>                                */
C1725 3
C1726 3 CC;
C1727 3 IF NOT NUMERIC$LIT THEN CALL INVALID$TYPE;
C1728 3 CALL SET$VALUE(NUMERIC$LITERAL);
C1729 3 CALL SET$VALUE2(STORE$CCONSTANT);
C1730 3 END;
C1731 3
C1732 3 /*      90          ZERO                                */
C1733 3
C1734 3 CALL SET$VALUE(LIT$ZERO);
C1735 3
C1736 3 /*      91      <LIT/ID> ::= <L/ID>                                */
C1737 3
C1738 3 ; /* NO ACTION REQUIRED */
C1739 3
C1740 3 /*      92          <NN-LIT>                                */
C1741 3
C1742 3 ; /* NO ACTION REQUIRED */
C1743 3
C1744 3 /*      93      <CFT-LIT/ID> ::= <LIT/ID>                                */
C1745 3
C1746 3 ; /* NO ACTION REQUIRED */
C1747 3
C1748 3 /*      94          <EMPTY>                                */
C1749 3
C1750 3 ; /* NO ACTION REQUIRED */
C1751 3
C1752 3 /*      95      <PROGRAM-ID> ::= <ID>                                */
C1753 3
C1754 3 CALL NCT$IMPLIMENTED; /* INTER PROG COMM */
C1755 3
C1756 3 /*      96                                */
C1757 3 ; /* NO ACTION REQUIRED */
C1758 3
C1759 3 /*      97      <PEAD-ID> ::= READ <ID>                                */
C1760 3
C1761 3 CALL READ$WRITE(0);
C1762 3
C1763 3 END; /* END OF CASE STATEMENT */
C1764 2 END CCCE$GEN;
C1765 1
C1766 1 GETIN1: PROCEDURE BYTE;
C1767 2 RETURN INDEX1(STATE);
C1768 2 ENC GETIN1;
C1769 1
C1770 1 GETIN2: PROCEDURE BYTE;
C1771 2 RETURN INDEX2(STATE);
C1772 2 ENC GETIN2;
C1773 1
C1774 1 INCSP: PROCEDURE;
C1775 2 VALUE(SF:=SF + 1)=0; /* CLEAR THE STACK WHILE INCREMENTING */
C1776 2 VALUE2(SF)=0;
C1777 2 IF SP>= PSTACKSIZE THEN CALL FATAL$ERROR('SC');
C1778 2 ENC INCSP;
C1779 1
C1780 1 LOCKAHEAD: PROCEDURE;
C1781 2 IF NCLCK THEN
C1782 2 CC;
C1783 2
C1784 2 CALL SCANNER;
C1785 2 NCLCK=FALSE;
C1786 2 IF PRINT$TOKEN THEN
C1787 2 DC;
C1788 2 CALL CRLF;
C1789 2 CALL PRINT$NUMBER(TOKEN);
C1790 2 CALL PRINT$CHAR(' ');
C1791 2 CALL PRINT$ACCUM;
C1792 2 ENC;
C1793 2 END LOCKAHEAD;
C1794 1
C1795 1 NC$CCNFLECT: PROCEDURE (CSTATE) BYTE;
C1796 2 DECLARE (CSTATE, I, J, K) BYTE;
C1797 2 J=INDEX1(CSTATE);
C1798 2 K=J + INDEX2(CSTATE) - 1;
C1799 2 DO I=J TO K;
C1800 2 IF READ1(I)=TOKEN THEN RETURN TRUE;
C1801 2 END;
C1802 2 RETURN FALSE;
C1803 2 END NC$CCNFLECT;
C1804 1
C1805 1 RECOVER: PROCEDURE BYTE;
C1806 2 DECLARE TSP BYTE, RSTATE BYTE;
C1807 2 DO FOREVER;
C1808 2 TSP=SP;
C1809 2 DO WHILE TSP <> 255;
C1810 2 IF NC$CCNFLECT(RSTATE:=STATESTACK(TSP)) THEN
C1811 2 CC; /* STATE WILL READ TOKEN */
C1812 2 IF SP<>TSP THEN SP = TSP - 1;
C1813 2 RETURN RSTATE;
C1814 2 END;
C1815 2 TSP = TSP - 1;
C1816 2 ENC;
C1817 2 CALL SCANNER; /* TRY ANOTHER TOKEN */
C1818 2 END;
C1819 2 END RECOVER;
C1820 1

```



```

01821 1      /* * * * * PROGRAM EXECUTION STARTS HERE * * */
01822 1
01823 1
01824 1
01825 1
01826 1      /* INITIALIZATION */
01827 1      TCKEN=62;      /* PRIME THE SCANNER WITH -PROCEDURE- */
01828 1      CALL MCVE(PASS1, TOP-PASS1$LEN, .OUTPUT$FCB, PASS1$LEN);
01829 1      /* THIS SETS
01830 1      OUTPUT FILE CCTRCL BLOCK
01831 1      TCGELES
01832 1      READ PCINTER
01833 1      NEXT SYMBOL TABLE POINTER
01834 1
01835 1      OUTPUT$END=(CUTPLT$PTR=.CUTPUT$BUFF-1)+128;
01836 1
01837 1      /* * * * * * * * * * * * * * * * * * * * *
01838 1
01839 1      DO WHILE COMPILEING;
01840 1      IF STATE <= MAXRNO THEN      /* READ STATE */
01841 1      CC;
01842 1          CALL INCSPI;
01843 1          STATESTACK(SP) = STATE; /* SAVE CURRENT STATE */
01844 1          CALL LCKKAHEAD;
01845 1          I=GETIN1;
01846 1          J = I + GETIN2 - 1;
01847 1          DO I=1 TO J;
01848 1              IF READ1(I) = TOKEN THEN
01849 1              DC;
01850 1              /* COPY THE ACCUMULATOR IF IT IS AN INPUT
01851 1              STRING. IF IT IS A RESERVED WORD IT DOES
01852 1              NOT NEED TO BE COPIED. */
01853 1              IF (TOKEN=INPUT$STR) OR (TOKEN=LITERAL) THEN
01854 1                  DO K=0 TO ACCUM;
01855 1                      VARC(K)=ACCUM(K);
01856 1                  END;
01857 1                  STATE=READ2(I);
01858 1                  NOLOCK=TRUE;
01859 1                  I=J;
01860 1              ELSE
01861 1              IF I=J THEN
01862 1              CC;
01863 1              CALL PRINT$ERROR('NP');
01864 1              CALL PRINT(, 'ERROR NEAR $');
01865 1              CALL PRINT$ACCUM;
01866 1              IF (STATE:=RECOVER)=0 THEN COMPILEING=FALSE;
01867 1              END;
01868 1          END;      /* END OF READ STATE */
01869 1      ELSE
01870 1      IF STATE>MAXRNO THEN      /* APPLY PRODUCTION STATE */
01871 1      CC;
01872 1          MP=SP - GETIN2;
01873 1          MPFI=MP + 1;
01874 1          CALL CCCE$GEN(STATE - MAXRNO);
01875 1          SP=MP;
01876 1          I=GETIN1;
01877 1          J=STATESTACK(SP);
01878 1          DO WHILE (K:=APPLY1(I)) <> 0 AND J<>K;
01879 1              I=I + 1;
01880 1              IF (K:=APPLY2(I))=0 THEN COMPILEING=FALSE;
01881 1              STATE=K;
01882 1          END;
01883 1      ELSE
01884 1      IF STATE<=MAXLNO THEN      /* LOCKAHEAD STATE */
01885 1      CC;
01886 1          I=GETIN1;
01887 1          CALL LCKKAHEAD;
01888 1          DO WHILE (K:=LOCK1(I))<>0 AND TOKEN <>K;
01889 1              I=I+1;
01890 1          END;
01891 1          STATE=LOCK2(I);
01892 1      END;
01893 1      /* PUSH STATES */
01894 1      CALL INCSPI;
01895 1      STATESTACK(SP)=GETIN2;
01896 1      STATE=GETIN1;
01897 1      END;      /* OF WHILE COMPILEING */
01898 1      CALL BYTES$CLT(ITER);
01899 1      DO WHILE CUTPUT$PTR<>.CUTPUT$BUFF;
01900 1          CALL BYTES$CLT(ITER);
01901 1      END;
01902 1      CALL CLCSE;
01903 1      CALL CRLF;
01904 1      CALL PRINT(, 'END OF PART 2 $');
01905 1      GO TO ECOT;
01906 1      EOF

```



```

000002 1      /*          CCBOL INTERPRETER          */
000003 1
000004 1  IOCH:      /* LOAC PCINT */
000005 1
000006 1      /* GLOBAL DECLARATIONS AND LITERALS */
000007 1
000008 1  DECLARE
000009 1
000010 1      LIT          LITERALLY          'LITERALLY',
000011 1  BDGS          LIT          '5H',          /* ENTRY TO OPERATING SYSTEM */
000012 1  CCCT          LIT          '0',
000013 1  CCCT          LIT          '13',
000014 1  F          LIT          '10',
000015 1  TRUE          LIT          '1',
000016 1  FALSE          LIT          '0',
000017 1  FOREVER          LIT          'WHILE TRUE';
000018 1
000019 1      /* UTILITY VARIABLES */
000020 1
000021 1  DECLARE
000022 1
000023 1  INDEX          BYTE,
000024 1  ASCTR          ADDRESS,
000025 1  CTR          BYTE,
000026 1  BASE          ADDRESS,
000027 1  B$BYTE          BASED BASE          BYTE,
000028 1  B$ACCR          BASED BASE          ADDRESS,
000029 1  HCLD          ADDRESS,
000030 1  H$BYTE          BASED HCLD          BYTE,
000031 1  H$ACCR          BASED HCLD          ADDRESS,
000032 1
000033 1      /* CCDE PCINTERS */
000034 1
000035 1  CODE$START          LIT          '2000H',
000036 1  PROGRAM$CCUNTEP          ADDRESS,
000037 1  C$BYTE          BASED PROGRAM$COUNTER          BYTE,
000038 1  C$ACCR          BASED PROGRAM$COUNTER          ADDRESS;
000039 1
000040 1
000041 1      /* * * * * * GLOBAL INPUT AND OUTPUT ROUTINES * * * * */
000042 1
000043 1  DECLARE
000044 1  CURRENT$PCB ADDRESS,
000045 1  START$CFFSET          LIT          '36';
000046 1
000047 1  MCN1: PROCEDURE (F,A);
000048 1      DECLARE F BYTE, A ADDRESS;
000049 1      GC TC BCCS;
000050 2  END MCN1;
000051 2
000052 1  MCN2: PROCEDURE (F,A)BYTE;
000053 1      DECLARE F BYTE, A ADDRESS;
000054 2  GC TC BCCS;
000055 2  END MCN2;
000056 2
000057 1  PRINT$CHAR: PROCEDURE (CHAR);
000058 1      DECLARE CHAR BYTE;
000059 2  CALL MCN1 (2,CHAR);
000060 2  END PRINT$CHAR;
000061 2
000062 1  CRLF: PROCEDURE;
000063 1      CALL PRINT$CHAR(CR);
000064 2  CALL PRINT$CHAR(LF);
000065 2  END CRLF;
000066 2
000067 1  PRINT: PROCEDURE (A);
000068 1      DECLARE A ADDRESS;
000069 2  CALL CRLF;
000070 2  CALL MCN1 (9,A);
000071 2  END PRINT;
000072 2
000073 1  READ: PROCEDURE (A);
000074 1      DECLARE A ADDRESS;
000075 2  CALL MCN1 (10,A);
000076 2  END READ;
000077 2
000078 1  PRINT$ERROR: PROCEDURE (CODE);
000079 1      DECLARE CODE ADDRESS;
000080 2  CALL CRLF;
000081 2  CALL PRINT$CHAR(HIGH(CODE));
000082 2  CALL PRINT$CHAR(LCW(CODE));
000083 2  END PRINT$ERROR;
000084 2
000085 1  FATAL$ERROR: PROCEDURE (CODE);
000086 1      DECLARE CODE ADDRESS;
000087 2  CALL PRINT$ERROR(CODE);
000088 2  CALL TIME(10);
000089 2  /* DEBUG
000090 2  CC TC BCCT;
000091 2  DEBUG */
000092 2  END FATAL$ERROR;
000093 2
000094 1  OPEN: PROCEDURE (ACCR) BYTE;
000095 1      DECLARE ACCR ADDRESS;
000096 2  RETURN MCN2 (15,ACCR);
000097 2  END OPEN;
000098 2
000099 1  CLCSE: PROCEDURE (ACCR);
000100 1      DECLARE ACCR ADDRESS;
000101 2  IF MON2 (16,ACCR) < 0 THEN CALL FATAL$ERROR ('CL');
000102 2  END CLCSE;
000103 2
000104 1
000105 1
000106 1
000107 1
000108 1
000109 1
000110 1

```



```

00111 1
00112 1 DELETE: PROCEDURE;
00113 2 CALL MCN1(15,CURRENT$FCB);
00114 2 END DELETE;
00115 1
00116 1
00117 1 MAKE: PROCEDURE (ADDR);
00118 2 DECLARE ADDR ADDRESS;
00119 2 IF MON2(22,ADDR)<>0 THEN CALL FATAL$ERROR('E');
00120 2 END MAKE;
00121 1
00122 1
00123 1 SET$DMA: PROCEDURE;
00124 2 CALL MCN1(26,CURRENT$FCB+ START$OFFSET);
00125 2 END SET$DMA;
00126 1
00127 1
00128 1 DISK$READ: PROCEDURE BYTE;
00129 2 RETURN MCN2(20,CURRENT$FCB);
00130 2 END DISK$READ;
00131 1
00132 1
00133 1 DISK$WRITE: PROCEDURE BYTE;
00134 2 RETURN MCN2(21,CURRENT$FCB);
00135 2 END DISK$WRITE;
00136 1
00137 1
00138 1
00139 1
00140 1
00141 1
00142 1
00143 1
00144 1
00145 1
00146 1
00147 1
00148 1
00149 1
00150 1
00151 1
00152 1
00153 1
00154 1
00155 1
00156 1
00157 1
00158 1
00159 1
00160 1
00161 1
00162 1
00163 1
00164 1
00165 1
00166 1
00167 1
00168 1
00169 1
00170 1
00171 1
00172 1
00173 1
00174 1
00175 1
00176 1
00177 1
00178 1
00179 1
00180 1
00181 1
00182 1
00183 1
00184 1
00185 1
00186 1
00187 1
00188 1
00189 1
00190 1
00191 1
00192 1
00193 1
00194 1
00195 1
00196 1
00197 1
00198 1
00199 1
00200 1
00201 2
00202 2
00203 2
00204 1
00205 1
00206 1
00207 2
00208 2
00209 2
00210 2
00211 1

DELETE: PROCEDURE;
CALL MCN1(15,CURRENT$FCB);
END DELETE;

MAKE: PROCEDURE (ADDR);
DECLARE ADDR ADDRESS;
IF MON2(22,ADDR)<>0 THEN CALL FATAL$ERROR('E');
END MAKE;

SET$DMA: PROCEDURE;
CALL MCN1(26,CURRENT$FCB+ START$OFFSET);
END SET$DMA;

DISK$READ: PROCEDURE BYTE;
RETURN MCN2(20,CURRENT$FCB);
END DISK$READ;

DISK$WRITE: PROCEDURE BYTE;
RETURN MCN2(21,CURRENT$FCB);
END DISK$WRITE;

/* * * * * * * * * * * UTILITY PROCEDURES * * * * * * * * * * */

DECLARE
SUBSCRIPT (8) ADDRESS;

RES: PROCEDURE (ADDR) ADDRESS;
/* THIS PROCEDURE RESOLVES THE ADDRESS OF A SUBSCRIPTED
IDENTIFIER OR A LITERAL CONSTANT */
DECLARE ADDR ADDRESS;
IF ADDR > 32 THEN RETURN ADDR;
IF ADDR < 9 THEN RETURN SUBSCRIPT(ADDR);
DO CASE ADDR - 9;
RETURN '0';
RETURN '1';
RETURN '2';
END;
RETURN C;
END RES;

MOVE: PROCEDURE (FROM, DESTINATION, COUNT);
DECLARE (FROM, DESTINATION, COUNT) ADDRESS;
IF BASED FROM, 0 BASED DESTINATION) BYTE;
DO WHILE (COUNT=COUNT - 1) <> 0FFFFH;
D=F;
FROM=FROM + 1;
DESTINATION=DESTINATION + 1;
END;
END MOVE;

FILL: PROCEDURE (DESTINATION, COUNT, CHAR);
DECLARE (DESTINATION, COUNT) ADDRESS;
(CHAR, 0 BASED DESTINATION) BYTE;
DO WHILE (COUNT=COUNT - 1) <> 0FFFFH;
D=CHAR;
DESTINATION=DESTINATION + 1;
END;
END FILL;

CCONVERT$TO$HEX: PROCEDURE (POINTER, COUNT) ADDRESS;
DECLARE FCINTER ADDRESS, COUNT BYTE;
A$CTR=0;
BASE=POINTER;
DO CTR = 0 TO COUNT;
A$CTR=SHL(A$CTR,3) + SHL(A$CTR,1) + B$EYTE(CTR) - '0';
END;
RETURN A$CTR;
END CCONVERT$TO$HEX;

/* * * * * * * * * * * CODE CONTRCL PROCEDURES * * * * * * * * * * */

DECLARE
BRANCH$FLAG BYTE INITIAL(TRUE);

INC$PTR: PROCEDURE (COUNT);
DECLARE COUNT BYTE;
PROGRAM$COUNTER=PROGRAM$COUNTER + COUNT;
END INC$PTR;

GET$CP$CODE: PROCEDURE BYTE;
CTR=C$EYTE;
CALL INC$PTR(1);
RETURN CTR;
END GET$CP$CODE;

```



```

00212 1 CCNDITIONAL$BRANCH: PROCEDURE(CCUNT);
00213 1 /* THIS PROCEDURE CONTROLS BRANCHING INSTRUCTIONS */
00214 2 DECLARE CCUNT BYTE;
00215 2 IF NOT BRANCH$FLAG THEN
00216 2 CC;
00217 2 BRANCH$FLAG=TRUE;
00218 3 PROGRAM$COUNTER=C$ADDR(CCUNT);
00219 3 END;
00220 3 ELSE CALL INC$PTR(SHL(CCUNT,1)+2);
00221 2 END CCNDITIONAL$BRANCH;
00222 1
00223 1 INCREMENT$OR$BRANCH: PROCEDURE(MARK);
00224 1 DECLARE MARK BYTE;
00225 2 IF MARK THEN CALL INC$PTR(2);
00226 2 ELSE PROGRAM$COUNTER=C$ADDR;
00227 2 END INCREMENT$CF$BRANCH;
00228 1
00229 1 /* * * * * * * * * * CCMPARISONS * * * * * * * * * */
00230 1
00231 1 CHAR$CCMPARE: PROCEDURE BYTE;
00232 1 BASE=C$ADDR;
00233 1 HCL=C$ADDR(1);
00234 1 DO A$CTR=1 TO C$ADDR(2)-1;
00235 1 IF B$BYTE(A$CTR) > H$BYTE(A$CTR) THEN RETURN 0;
00236 1 IF B$BYTE(A$CTR) < H$BYTE(A$CTR) THEN RETURN 1;
00237 1 END;
00238 1 RETURN 2;
00239 1 END CHAR$CCMPARE;
00240 1
00241 1 STRING$CCMPARE: PROCEDURE(PIVOT);
00242 1 DECLARE PIVOT BYTE;
00243 1 IF CHAR$CCMPARE<>PIVOT THEN BRANCH$FLAG=NOT BRANCH$FLAG;
00244 1 CALL CCNDITIONAL$BRANCH(3);
00245 1 END STRING$CCMPARE;
00246 1
00247 1 NUMERIC: PROCEDURE(CHAR) BYTE;
00248 1 DECLARE CHAR BYTE;
00249 1 RETURN (CHAR >='0') AND (CHAR <='9');
00250 1 END NUMERIC;
00251 1
00252 1 LETTER: PROCEDURE(CHAR) BYTE;
00253 1 DECLARE CHAR BYTE;
00254 1 RETURN (CHAR >='A') AND (CHAR <='Z');
00255 1 END LETTER;
00256 1
00257 1 SIGN: PROCEDURE(CHAR) BYTE;
00258 1 DECLARE CHAR BYTE;
00259 1 RETURN (CHAR='+') OR (CHAR='-');
00260 1 END SIGN;
00261 1
00262 1 CCMP$NUM$UNSIGNED: PROCEDURE;
00263 1 BASE=C$ADDR;
00264 1 CC A$CTR=0 TO C$ADDR(2)-1;
00265 1 IF NOT NUMERIC(B$BYTE(A$CTR)) THEN
00266 1 DO;
00267 1 BRANCH$FLAG=NOT BRANCH$FLAG;
00268 1 RETURN;
00269 1 END;
00270 1 END;
00271 1 CALL CCNDITIONAL$BRANCH(2);
00272 1 END CCMP$NUM$UNSIGNED;
00273 1
00274 1 CCMP$NUM$SIGN: PROCEDURE;
00275 1 BASE=C$ADDR;
00276 1 CC A$CTR=0 TO C$ADDR(2)-1;
00277 1 IF NOT (NUMERIC(CTR=B$BYTE(A$CTR))
00278 1 OR SIGN(CTR)) THEN
00279 1 DO;
00280 1 BRANCH$FLAG=NOT BRANCH$FLAG;
00281 1 RETURN;
00282 1 END;
00283 1 END;
00284 1 CALL CCNDITIONAL$BRANCH(2);
00285 1 END CCMP$NUM$SIGN;
00286 1
00287 1 CCMP$ALPHA: PROCEDURE;
00288 1 BASE=C$ADDR;
00289 1 CC A$CTR=0 TO C$ADDR(2)-1;
00290 1 IF NOT (LETTER(B$BYTE(A$CTR))
00291 1 OR SIGN(CTR)) THEN
00292 1 DO;
00293 1 BRANCH$FLAG=NOT BRANCH$FLAG;
00294 1 RETURN;
00295 1 END;
00296 1 END;
00297 1 CALL CCNDITIONAL$BRANCH(2);
00298 1 END CCMP$ALPHA;
00299 1
00300 1
00301 1
00302 1
00303 1
00304 1
00305 1
00306 1
00307 1
00308 1
00309 1
00310 1
00311 1

```



```

00312 1
00313 1 / * * * * * * * * * NUMERIC OPERATIONS * * * * * * * * * /
00314 1
00315 1
00316 1
00317 1
00318 1 DECLARE
00319 1 (RC,R1,R2) (10) BYTE, /* REGISTERS */
00320 1 (SIGNC,SIGN1,SIGN2) BYTE,
00321 1 (CEC$PT0,DEC$PT1,DEC$PT2) BYTE,
00322 1 CVERFLW BYTE,
00323 1 P$PTR BYTE,
00324 1 SWITCH BYTE,
00325 1 SIGNIF$NO BYTE,
00326 1 ZERO$RESULT BYTE,
00327 1 ZCNE LIT '10H',
00328 1 PCSITIVE LIT '1',
00329 1 NEGITIVE LIT '0';
00330 1
00331 1 CHECK$FCR$SIGN: PROCEDURE (CHAR) BYTE;
00332 1 DECLARE CHAR BYTE;
00333 1 IF NUMERIC (CHAR) THEN RETURN POSITIVE;
00334 1 IF NUMERIC (CHAR - ZONE) THEN RETURN NEGATIVE;
00335 1 CALL PRINT$ERROR ('SI');
00336 1 RETURN PCSITIVE;
00337 1 END CHECK$FCR$SIGN;
00338 1
00339 1
00340 1 STORE$IMMEDIATE: PROCEDURE;
00341 1 CC CTR=0 TC 5;
00342 1 RO (CTR)=R2 (CTR);
00343 1 END;
00344 1 CEC$PT0=DEC$PT2;
00345 1 SIGNC=SIGN2;
00346 1 END STORE$IMMEDIATE;
00347 1
00348 1
00349 1 CNE$LEFT: PROCEDURE;
00350 1 DECLARE FLAG BYTE;
00351 1 IF ((FLAG=SHR (B$BYTE,4))=0) OR (FLAG=9) THEN
00352 1 CC;
00353 1 OC CTR=C TO 8;
00354 1 B$BYTE (CTR)=SHL (B$BYTE (CTR),4) OR SHR (B$BYTE (CTR + 1),4);
00355 1 END;
00356 1 B$BYTE (5)=SHL (B$BYTE (9),4) OR FLAG;
00357 1 END;
00358 1 ELSE CVERFLW=TRUE;
00359 1 END CNE$LEFT;
00360 1
00361 1
00362 1 CNE$RIGHT: PROCEDURE;
00363 1 CTR=10;
00364 1 CC INDEX=1 TC 9;
00365 1 CTR=CTR-1;
00366 1 B$BYTE (CTR)=SHR (B$BYTE (CTR),4) OR SHL (B$BYTE (CTR-1),4);
00367 1 END;
00368 1 B$BYTE=SHR (B$BYTE,4);
00369 1 END CNE$RIGHT;
00370 1
00371 1
00372 1 SHIFT$RIGHT: PROCEDURE (CCUNT);
00373 1 DECLARE CCUNT BYTE;
00374 1 CC CTR=1 TC CCUNT;
00375 1 CALL CNE$RIGHT;
00376 1 END;
00377 1 END SHIFT$RIGHT;
00378 1
00379 1
00380 1 SHIFT$LEFT: PROCEDURE (CCUNT);
00381 1 DECLARE CCUNT BYTE;
00382 1 CVERFLW=FALSE;
00383 1 CC CTR=1 TC CCUNT;
00384 1 CALL CNE$LEFT;
00385 1 IF CVERFLW THEN RETURN;
00386 1 END;
00387 1 END SHIFT$LEFT;
00388 1
00389 1
00390 1 ALLIGN: PROCEDURE;
00391 1 BASE=RC;
00392 1 IF DEC$PT0 > DEC$PT1 THEN CALL SHIFT$RIGHT (CEC$PT0-DEC$PT1);
00393 1 ELSE CALL SHIFT$LEFT (DEC$PT1-DEC$PT0);
00394 1 END ALLIGN;
00395 1

```



```

00356 1
00357 1
00358 1
00359 1
00360 1
00361 1
00362 1
00363 1
00364 1
00365 1
00366 1
00367 1
00368 1
00369 1
00370 1
00371 1
00372 1
00373 1
00374 1
00375 1
00376 1
00377 1
00378 1
00379 1
00380 1
00381 1
00382 1
00383 1
00384 1
00385 1
00386 1
00387 1
00388 1
00389 1
00390 1
00391 1
00392 1
00393 1
00394 1
00395 1
00396 1
00397 1
00398 1
00399 1
00400 1
00401 1
00402 1
00403 1
00404 1
00405 1
00406 1
00407 1
00408 1
00409 1
00410 1
00411 1
00412 1
00413 1
00414 1
00415 1
00416 1
00417 1
00418 1
00419 1
00420 1
00421 1
00422 1
00423 1
00424 1
00425 1
00426 1
00427 1
00428 1
00429 1
00430 1
00431 1
00432 1
00433 1
00434 1
00435 1
00436 1
00437 1
00438 1
00439 1
00440 1
00441 1
00442 1
00443 1
00444 1
00445 1
00446 1
00447 1
00448 1
00449 1
00450 1
00451 1
00452 1
00453 1
00454 1
00455 1
00456 1
00457 1
00458 1
00459 1
00460 1
00461 1
00462 1
00463 1
00464 1
00465 1
00466 1
00467 1
00468 1
00469 1
00470 1
00471 1
00472 1
00473 1
00474 1
00475 1
00476 1
00477 1
00478 1
00479 1
00480 1
00481 1
00482 1
00483 1
00484 1
00485 1
00486 1
00487 1
00488 1
00489 1
00490 1
00491 1
00492 1
00493 1
00494 1
00495 1
00496 1
00497 1
00498 1
00499 1
00500 1
00501 1

ADD$RC: PROCEDURE(SECCNT, DEST);
DECLARE (SECCNT, DEST) ADDRESS, (CY,A,8,1) BYTE;
HOLD= SECCNT;
BASE= DEST;
CY=0;
CTR=9;
CC INDEX=1 TC 10;
A=RC(CTR);
B=H$BYTE(CTR);
I=DEC(A+CY);
CY=CARRY;
I=DEC(I+B);
CY=(CY OR CARRY) AND 1;
B$BYTE(CTR)=I;
CTR=CTR-1;
END;
IF CY THEN
CC:
CTR=9;
OD INDEX=1 TO 10;
I=2(CTR);
I=DEC(I+CY);
CY=CARRY AND 1;
R2(CTR)=I;
CTR=CTR-1;
END;
END;
END ADD$RO;

COMPLIMENT: FPCEDURE(NUMB);
DECLARE NUMB BYTE;
CC CASE NUMB;
HOLD=.R1;
HOLD=.R2;
HOLD=.R2;
END;
IF SIGN(C(NUMB)) THEN SIGNO(NUMB) = NEGATIVE;
ELSE SIGNO(NUMB)= POSITIVE;
CC CTR=0 TC 9;
H$BYTE(CTR)=99H - H$BYTE(CTR);
END;
END COMPLIMENT;

CHECK$RESULT: FPCEDURE;
IF SHR(R2,4)=9 THEN CALL COMPLIMENT(2);
IF SHR(R2,4)<>0 THEN OVERFLW=TRUE;
END CHECK$RESULT;

CHECK$SIGN: FPCEDURE;
IF SIGNO AND SIGN1 THEN
CC:
SIGN2=POSITIVE;
RETURN;
END;
SIGN2=NEGATIVE;
IF NOT SIGNO AND NOT SIGN1 THEN RETURN;
IF SIGNO THEN CALL COMPLIMENT(1);
ELSE CALL COMPLIMENT(0);
END CHECK$SIGN;

LEADING$ZERCE: FPCEDURE (ADDR) BYTE;
DECLARE CCLNT BYTE, ADDR ADDRESS;
CCLNT=0;
BASE=ADDR;
CC CTR=0 TC 9;
IF (B$BYTE(CTR) AND OFOH) <> 0 THEN RETURN COUNT;
CCLNT=CCLNT + 1;
IF (B$BYTE(CTR) AND OFH) <> 0 THEN RETURN COUNT;
CCLNT=CCLNT + 1;
END;
RETURN CCLNT;
END LEADING$ZERCE;

CHECK$DECIMAL: FPCEDURE;
IF DEC$PT2<>(CTR=C$BYTE(3)) THEN
CC:
BASE=.R2;
IF DEC$PT2 > CTR THEN CALL SHIFT$RIGHT(DEC$PT2-CTR);
ELSE CALL SHIFT$LEFT(CTR-DEC$PT2);
END;
IF LEADING$ZERCE(.R2) < 19 - C$BYTE(2) THEN OVERFLOW = TRUE;
END CHECK$DECIMAL;

ADD: FPCEDURE;
OVERFLW=FALSE;
CALL ALIGN;
CALL CHECK$SIGN;
CALL ADDR0(.R1,.R2);
CALL CHECK$RESULT;
END ADD;

ADD$SEPIES: FPCEDURE(COUNT);
DECLARE (I,COUNT) BYTE;
CC I=1 TO CCLNT;
CALL ADD$RO(.R2,.R2);
END;
END ADD$SERIES;

```



```

00502 1
00503 1 SET$MLLT$CIV: PROCEDURE;
00504 1 CVERFLC=FALSE;
00505 2 IF (SIGNC AND SIGN1) OR
00506 2 (NOT SIGNC AND NOT SIGN1) THEN SIGN2=POSITIVE;
00507 2 ELSE SIGN2=NEGATIVE;
00508 2 CALL FILL(.R2,10,0);
00509 2 ENC SET$MULT$CIV;
00510 1
00511 1
00512 1 R1$GREATER: PROCEDURE BYTE;
00513 2 DECLARE I BYTE;
00514 2 CC CTR=0 TO 9;
00515 2 IF R1(CTR)>(I:=99H-R0(CTR)) THEN RETURN TRUE;
00516 2 IF R1(CTR)<I THEN RETURN FALSE;
00517 3 ENC;
00518 2 RETURN TRUE;
00519 2 END R1$GREATER;
00520 1
00521 1
00522 1 MULTIPLY: PROCEDURE (VALUE);
00523 2 DECLARE VALUE BYTE;
00524 2 IF VALUE<0 THEN CALL ADD$SERIES(VALUE);
00525 2 BASE=.RC;
00526 2 CALL CNE$LEFT;
00527 2 END MULTIPLY;
00528 1
00529 1
00530 1 DIVIDE: PROCEDURE;
00531 2 DECLARE (I,J,K,LZ0,LZ1) BYTE;
00532 2 CALL SET$MLLT$CIV;
00533 2 IF (LZ0:=LEADING$ZEROCES(BASE:=.R0))<>
00534 2 (LZ1:=LEADING$ZEROCES(.R1)) THEN
00535 2 CC;
00536 3 IF LZ0>LZ1 THEN
00537 3 DO;
00538 4 CALL SHIFT$LEFT(I:=LZ0-LZ1);
00539 4 DEC$PTO=DEC$PTO + I;
00540 4 ENC;
00541 3 ELSE CC;
00542 3 CALL SHIFT$RIGHT(I:=LZ1-LZ0);
00543 3 DEC$PTO=DEC$PTO - I;
00544 3 ENC;
00545 3 ENC;
00546 2 DECPT2= 20 - LZ1 + DECPTO - DECPT1;
00547 2 CALL COMPLEMENT(0);
00548 2 DO I=LZ1 TO 15;
00549 3 J=0;
00550 3 OC WHILE R1$GREATER;
00551 3 CALL ADD$RO(.R1,.R1);
00552 3 J=J+1;
00553 3 ENC;
00554 3 K=SHR(1,1);
00555 3 IF 1 THEN R2(K)=R2(K) OR J;
00556 3 ELSE R2(K)=R2(K) OR SHL(J,4);
00557 2 ENC;
00558 2 END DIVIDE;
00559 1
00560 1
00561 1
00562 1
00563 1 LOAD$SCHAR: PROCEDURE (CHAR);
00564 2 DECLARE CHAR BYTE;
00565 2 IF (SWITCH:=NOT SWITCH) THEN
00566 2 B$BYTE(R$PTR)=B$BYTE(R$PTR) OR SHL(CHAR - 30H,4);
00567 2 ELSE B$BYTE(R$PTR:=R$PTR-1)=CHAR - 30H;
00568 2 END LOAD$SCHAR;
00569 1
00570 1
00571 1 LOAD$NLMBERS: PROCEDURE (ADDR,CNT);
00572 2 DECLARE ADDR ADDRESS, (1,CNT) BYTE;
00573 2 FCLD=RES(ADDR);
00574 2 CTR=CNT;
00575 2 DO INDEX = 1 TO CNT;
00576 2 CTR=CTR-1;
00577 2 CALL LOAD$SCHAR(H$BYTE(CTR));
00578 2 ENC;
00579 2 CALL INC$PTR(5);
00580 2 END LOAD$NLMBERS;
00581 1
00582 1
00583 1 SET$LCAD: PROCEDURE (SIGN$IN);
00584 2 DECLARE SIGN$IN BYTE;
00585 2 CC CASE (CTR:=C$BYTE(4));
00586 3 BASE=.RC;
00587 3 BASE=.R1;
00588 3 BASE=.R2;
00589 3 ENC;
00590 2 DEC$PTO(CTR)=C$BYTE(3);
00591 2 SIGNO(CTR)=SIGN$IN;
00592 2 CALL FILL (BASE,10,0);
00593 2 R$PTR=9;
00594 2 SWITCH=FALSE;
00595 2 END SET$LCAD;
00596 1
00597 1
00598 1 LOAD$NUMERIC: PROCEDURE;
00599 2 CALL SET$LCAD(1);
00600 2 CALL LOAD$NLMBERS(C$ADDR,C$BYTE(2));
00601 2 END LOAD$NUMERIC;
00602 1

```



```

00603 1
00604 1 LOAD$NUM$LIT: PROCEDURE;
00605 2 DECLARE(LIT$SIZE,FLAG) BYTE;
00606 2
00607 2 CHAR$SIGN: PROCEDURE;
00608 3 LIT$SIZE=LIT$SIZE - 1;
00609 3 HOLD=HOLD + 1;
00610 3 END CHAR$SIGN;
00611 2
00612 2 LIT$SIZE=C$BYTE(2);
00613 2 HOLD=C$ADDR;
00614 2 IF H$BYTE='-' THEN
00615 2 CC;
00616 2 CALL CHAR$SIGN;
00617 2 CALL SET$LOAD(NEGATIVE);
00618 2
00619 2 ELSE CC;
00620 2 IF H$BYTE='+' THEN CALL CHAR$SIGN;
00621 2 CALL SET$LOAD(POSITIVE);
00622 2
00623 2 END;
00624 2 FLAG=0;
00625 2 CTR=LIT$SIZE;
00626 2 DO INDEX=1 TO LIT$SIZE;
00627 2 CTR=CTR-1;
00628 2 IF H$BYTE(CTR)='.' THEN FLAG=LIT$SIZE - (CTR+1);
00629 2 ELSE CALL LOAD$AS$CHAR(H$BYTE(CTR));
00630 2
00631 2 EEC$PTO(C$BYTE(4))= FLAG;
00632 2 CALL INC$PTR(5);
00633 2 END LOAD$NUM$LIT;
00634 1
00635 1 STCRE$CNE: PROCEDURE;
00636 1 IF (SWITCH:=ACT SWITCH) THEN
00637 1 B$BYTE=SFR(H$BYTE,4) OR '0';
00638 1 ELSE CC;
00639 1 HOLD=HOLD-1;
00640 1 B$BYTE=(H$BYTE AND OFH) OR '0';
00641 1
00642 1 END;
00643 1 END STCRE$CNE;
00644 1
00645 1 STCRE$AS$CHAR: PROCEDURE(COUNT);
00646 1 DECLARE COUNT BYTE;
00647 1 SWITCH=FALSE;
00648 1 HOLD=HOLD + 1;
00649 1 DO CTR=1 TO COUNT;
00650 1 CALL STCRE$CNE;
00651 1
00652 1 END STCRE$AS$CHAR;
00653 1
00654 1 SET$ZONE: PROCEDURE (ADDR);
00655 1 DECLARE ADDR ADDRESS;
00656 1 IF NOT SIGN2 THEN
00657 1 DO;
00658 1 BASE=ADDR;
00659 1 B$BYTE=B$BYTE OR ZONE;
00660 1
00661 1 END;
00662 1 CALL INC$PTR(4);
00663 1 END SET$ZONE;
00664 1
00665 1 SET$SIGN$SEP: PROCEDURE (ADDR);
00666 1 DECLARE ADDR ADDRESS;
00667 1 BASE=ADDR;
00668 1 IF SIGN2 THEN B$BYTE='+';
00669 1 ELSE B$BYTE='-';
00670 1 CALL INC$PTR(4);
00671 1 END SET$SIGN$SEP;
00672 1
00673 1 STCRE$NUMERIC: PROCEDURE;
00674 1 CALL CHECK$DECIMAL;
00675 1 BASE=C$ADDR + C$BYTE(2) -1;
00676 1 CALL STCRE$AS$CHAR(C$BYTE(2));
00677 1 END STCRE$NUMERIC;
00678 1
00679 1
00680 1
00681 1
00682 1
00683 1
00684 1
00685 1
00686 1 /* * * * * INPUT-OUTPUT ACTIONS * * * * * */
00687 1
00688 1
00689 1
00690 1 DECLARE
00691 1 FLAG$OFFSET LIT '33';
00692 1 EXTENT$OFFSET LIT '12';
00693 1 RECSAC LIT '32';
00694 1 FTR$OFFSET LIT '17';
00695 1 BUFF$LENGTH LIT '128';
00696 1 VAR$END LIT 'CR';
00697 1 TERMINATOR LIT '1AH';
00698 1 ENC$CF$RECCRC BYTE;
00699 1 INVALID BYTE;
00700 1 RANDCM$FILE BYTE;
00701 1 CLAREN$FLAG BYTE;
00702 1 FCB$BASE BASED CURRENT$FCB BYTE;
00703 1 FCB$ADDR BASED CURRENT$FCB ADDRESS;
00704 1 BUFF$PTR ADDRESS;
00705 1 BUFF$END ADDRESS;
00706 1 BUFF$START ADDRESS;
00707 1 BUFF$BYTE BASED BUFF$PTR BYTE;
00708 1 CON$BUFF ADDRESS INITIAL (80H);
00709 1 CON$BYTE BASED CON$BUFF BYTE;
00710 1 CON$INPUT ADDRESS INITIAL (82H);
00711 1

```



```

00712 1
00713 1 ACCEPT: PRCCEDURE;
00714 2 CALL CRLF;
00715 2 CALL PRINT$CHAR(3FH);
00716 2 CALL CRLF;
00717 2 CALL FILL(CCN$INPUT,(CON$BYTE:=C$BYTE(2)),' ');
00718 2 CALL READ(CCN$BUFF);
00719 2 CALL MOVE(CCN$INPLT,RES(C$ADDR),CON$BYTE);
00720 2 CALL INC$PTR(3);
00721 2 END ACCEPT;
00722 1
00723 1
00724 1 DISPLAY: PRCCEDURE;
00725 2 BASE=C$ADDR;
00726 2 CALL CRLF;
00727 2 CC CTR=0 TO C$BYTE(2)-1;
00728 2 CALL PRINT$CHAR(B$BYTE(CTR));
00729 2 END;
00730 2 CALL INC$PTR(3);
00731 2 END DISPLAY;
00732 1
00733 1
00734 1 SET$FILE$TYPE: PRCCEDURE(TYPE);
00735 2 CECLARE TYPE BYTE;
00736 2 BASE=C$ADDR;
00737 2 B$BYTE(FLAG$CFFSET)=TYPE;
00738 2 END SET$FILE$TYPE;
00739 1
00740 1
00741 1 GET$FILE$TYPE: PRCCEDURE BYTE;
00742 2 BASE=C$ADDR;
00743 2 RETURN B$BYTE(FLAG$CFFSET);
00744 2 END GET$FILE$TYPE;
00745 1
00746 1
00747 1 SET$ISC: PRCCEDURE;
00748 2 END$CF$RECCRC,INVALID=FALSE;
00749 2 IF C$ACCF=CURRENT$FCB THEN RETURN;
00750 2 /* STORE CURRENT PCINTERS AND SET INTERNAL WRITE MARK */
00751 2 BASE=CURRENT$FCB;
00752 2 FCB$ADDR(PTR$CFFSET)=BUFF$PTR;
00753 2 FCB$BYTE(FLAG$CFFSET)=CURRENT$FLAG;
00754 2 /* LOAD NEW VALUES */
00755 2 BLFF$END=(BLFF$START:=(CURRENT$FCB:=C$ADDR)+START$CFFSET)
00756 2 + BLFF$LENGTH;
00757 2 CURRENT$FLAG=FCB$BYTE(FLAG$CFFSET);
00758 2 BLFF$PTR=FCB$ADDR(PTR$CFFSET);
00759 2 END SET$ISC;
00760 1
00761 1
00762 1 OPEN$FILE: PRCCEDURE(TYPE);
00763 2 CECLARE TYPE BYTE;
00764 2 CALL SET$FILE$TYPE(TYPE);
00765 2 CTR=OPEN(CURRENT$FCB:=C$ADDR);
00766 2 DO CASE TYPE-1;
00767 2 /* INFLT */
00768 2 DO;
00769 2 IF CTR=255 THEN CALL PRINT$ERROR('NF');
00770 2 FCB$ADDR(PTR$DFFSET)=CURRENT$FCB+100H;
00771 2 END;
00772 2 /* CLTFLT */
00773 2 DO;
00774 2 CALL DELETE;
00775 2 CALL MAKE(C$ADDR);
00776 2 FCB$ADDR(PTR$OFFSET)=CURRENT$FCB+START$OFFSET-1;
00777 2 END;
00778 2 /* I-C */
00779 2 DO;
00780 2 IF CTR=255 THEN CALL FATAL$ERROR('NF');
00781 2 FCB$ADDR(PTR$GFFSET)=CURRENT$FCB + 100H;
00782 2 END;
00783 2 END;
00784 2 CURRENT$FCB=C; /* FORCE A PARAMETER LOAD */
00785 2 CALL SET$ISC;
00786 2 CALL INC$PTR(2);
00787 2 END OPEN$FILE;
00788 1
00789 1
00790 1 WRITE$MARK: PRCCEDURE BYTE;
00791 2 RETURN RCL(CURRENT$FLAG,1);
00792 2 END WRITE$MARK;
00793 1
00794 1
00795 1 SET$WRITE$MARK: PRCCEDURE;
00796 2 CURRENT$FLAG=CURRENT$FLAG OR 80H;
00797 2 END SET$WRITE$MARK;
00798 1
00799 1
00800 1 WRITE$RECORD: PRCCEDURE;
00801 2 IF NOT SFR(CURRENT$FLAG,1) THEN CALL FATAL$ERROR('WI');
00802 2 CALL SET$CMA;
00803 2 CURRENT$FLAG=CURRENT$FLAG AND 3FH;
00804 2 IF (CTR:=DISK$READ)=0 THEN RETURN;
00805 2 INVALID=TRUE;
00806 2 END WRITE$RECORD;
00807 1
00808 1
00809 1 READ$RECORD: PRCCEDURE;
00810 2 CALL SET$CMA;
00811 2 IF WRITE$MARK THEN CALL WRITE$RECORD;
00812 2 IF (CTR:=DISK$READ)=0 THEN RETURN;
00813 2 IF CTR=1 THEN END$OF$RECORD=TRUE;
00814 2 ELSE INVALID=TRUE;
00815 2 END READ$RECORD;
00816 1

```



```

00817 1
00818 1 READ$BYTE: PROCEDURE BYTE;
00819 2 IF (BUFF$PTR:=BUFF$PTR + 1) >= BUFF$END THEN
00820 2 CC;
00821 3 CALL READ$RECORD;
00822 3 IF ENC$CF$RECORD THEN RETURN TERMINATOR;
00823 3 BUFF$PTR=BUFF$START;
00824 3 END;
00825 2 RETURN ELFF$BYTE;
00826 1 END READ$BYTE;
00827 1
00828 1
00829 1 WRITE$BYTE: PROCEDURE (CHAR);
00830 2 DECLARE CHAR BYTE;
00831 2 IF (BUFF$PTR:=BUFF$PTR+1) >= BUFF$END THEN
00832 2 CC;
00833 2 CALL WRITE$RECORD;
00834 2 BUFF$PTR=BUFF$START;
00835 2 END;
00836 2 CALL SET$WRIT$MARK;
00837 2 ELFF$BYTE=CHAR;
00838 1 END WRITE$BYTE;
00839 1
00840 1
00841 1 WRITE$END$MARK: PROCEDURE;
00842 2 CALL WRITE$BYTE(CR);
00843 2 CALL WRITE$BYTE(LF);
00844 1 END WRITE$END$MARK;
00845 1
00846 1
00847 1 READ$END$MARK: PROCEDURE;
00848 2 IF READ$BYTE<>CR THEN CALL PRINT$ERROR('EM');
00849 2 IF READ$BYTE<>LF THEN CALL PRINT$ERROR('EM');
00850 1 END READ$END$MARK;
00851 1
00852 1 READ$VARIABLE: PROCEDURE;
00853 2 CALL SET$IC;
00854 2 BASE=C$ADDR(1);
00855 2 CC A$CTR=0 TC C$ADDR(2)-1;
00856 2 IF (CTR:=(B$BYTE(A$CTR):=READ$BYTE)) = VAR$END THEN
00857 2 DD;
00858 2 CTR=READ$BYTE;
00859 2 RETURN;
00860 2 END;
00861 2 IF CTR=TERMINATOR THEN
00862 2 CC;
00863 2 END$OF$RECORD=TRUE;
00864 2 RETURN;
00865 2 END;
00866 2 CALL READ$END$MARK;
00867 1 END READ$VARIABLE;
00868 1
00869 1
00870 1 WRITE$VARIABLE: PROCEDURE;
00871 2 DECLARE CCOUNT ADDRESS;
00872 2 CALL SET$IC;
00873 2 BASE=C$ADDR(1);
00874 2 CCOUNT=C$ADDR(2);
00875 2 DO WHILE (B$BYTE(CCOUNT:=COUNT-1)<>' ') AND (CCOUNT<>0);
00876 2 DD;
00877 2 CC A$CTR=0 TC CCOUNT;
00878 2 CALL WRITE$BYTE(B$BYTE(A$CTR));
00879 2 END;
00880 2 CALL WRITE$END$MARK;
00881 1 END WRITE$VARIABLE;
00882 1
00883 1
00884 1 READ$TO$MEMORY: PROCEDURE;
00885 2 CALL SET$IC;
00886 2 BASE=C$ADDR(1);
00887 2 CC A$CTR=0 TC C$ADDR(2)-1;
00888 2 IF (B$BYTE(A$CTR):=READ$BYTE)=TERMINATOR THEN
00889 2 CC;
00890 2 END$OF$RECORD=TRUE;
00891 2 RETURN;
00892 2 END;
00893 2 CALL READ$END$MARK;
00894 1 END READ$TO$MEMORY;
00895 1
00896 1
00897 1 WRITE$FROM$MEMORY: PROCEDURE;
00898 2 CALL SET$IC;
00899 2 BASE=C$ADDR(1);
00900 2 CC A$CTR=0 TC C$ADDR(2)-1;
00901 2 CALL WRITE$BYTE(B$BYTE(A$CTR));
00902 2 END;
00903 2 CALL WRITE$END$MARK;
00904 1 END WRITE$FROM$MEMORY;
00905 1
00906 1
00907 1
00908 1

```



```

C1018 1 CHECK$EDIT: PROCEDURE(CHAR);
C1019 1 DECLARE CHAR BYTE;
C1020 2 IF (CHAR='0') OR (CHAR='/') THEN CALL INC$PCLO;
C1021 2 IF CHAR='B' THEN
C1022 2 OC;
C1023 2 H$BYTE=' ';
C1024 2 CALL INC$HCLO;
C1025 3
C1026 3 END;
C1027 2 ELSE IF CHAR='A' THEN
C1028 2 OC;
C1029 2 IF NOT LETTER(B$BYTE) THEN CALL PRINT$ERROR('IC');
C1030 3 CALL LCAD$INC;
C1031 3
C1032 2 END;
C1033 2 ELSE IF CHAR='9' THEN
C1034 2 OC;
C1035 2 IF NOT NUMERIC(B$BYTE) THEN CALL PRINT$ERROR('IC');
C1036 3 CALL LCAD$INC;
C1037 3
C1038 2 END;
C1039 2 ELSE CALL LCAD$INC;
C1040 2 END CHECK$EDIT;
C1041 1
C1042 1 /* * * * * * MACHINE ACTIONS * * * * * */
C1043 1
C1044 1 STCP: PROCEDURE;
C1045 2 CALL PRINT('EOF $');
C1046 2 CC TC BCCT;
C1047 2 END STCP;
C1048 1
C1049 1 /* * * * * *
C1050 1
C1051 1 THE PROCEDURE BELOW CONTROLS THE EXECUTION OF THE CODE.
C1052 1 IT DECODES EACH OP-CODE AND PERFORMS THE ACTIONS
C1053 1
C1054 1 * * * * * */
C1055 1
C1056 1 EXECUTE: PROCEDURE;
C1057 1 CC FOREVER;
C1058 2 DO CASE GET$CP$CODE;
C1059 2
C1060 2 ; /* CASE ZERO NOT USED */
C1061 4
C1062 4 /* ADD */
C1063 4
C1064 4 CALL ADD;
C1065 4
C1066 4 /* SUB */
C1067 4
C1068 4 CC;
C1069 4 CALL COMPLIMENT(0);
C1070 4 IF SIGN THEN SIGN=NEGATIVE;
C1071 4 ELSE SIGN=POSITIVE;
C1072 4 CALL ADD;
C1073 4
C1074 4 END;
C1075 4
C1076 4 /* MLL */
C1077 4
C1078 4 CC;
C1079 4 DECLARE I BYTE;
C1080 4 CALL SET$MULT$DIV;
C1081 4 DECPT1,DECPT2=DECPT1 + DECPTC;
C1082 4 CALL ALIGN;
C1083 4 CALL MULTIPLY(SHR(R1(I:=9),4));
C1084 4 DO INDEX=1 TC 9;
C1085 4 CALL MULTIPLY(R1(I:=I-1) AND OFH);
C1086 4 CALL MULTIPLY(SHR(R1(I),4));
C1087 4
C1088 4 END;
C1089 4
C1090 4 /* DIV */
C1091 4
C1092 4 CALL DIVIDE;
C1093 4
C1094 4 /* NEG */
C1095 4
C1096 4 BRANCH$FLAG=FALSE;
C1097 4
C1098 4 /* STP */
C1099 4
C1100 4 CALL STCP;
C1101 4
C1102 4 /* STI */
C1103 4
C1104 4 CALL STORE$IMMEDIATE;
C1105 4
C1106 4 /* RND */
C1107 4
C1108 4 CC;
C1109 4 CALL STORE$IMMEDIATE;
C1110 4 CALL FILL(.R2,10,0);
C1111 4 R2(9)=1;
C1112 4 CALL ADD;
C1113 4
C1114 4 END;
C1115 4
C1116 4 /* RET */
C1117 4
C1118 4 CC;
C1119 4 IF C$ADDR<>0 THEN
C1120 4 DO;
C1121 4 A$CTR=C$ADDR;
C1122 4 C$ADDR=0;
C1123 4 PROGRAM$COUNTER=A$CTR;
C1124 4
C1125 4 END;
C1126 4 ELSE CALL INC$PTR(2);
C1127 4
C1128 4 END;

```



```

C1126 4      /* CLS */
C1127 4
C1128 4
C1129 4      DC;
C1130 5      CALL SET$I$0;
C1131 5      IF WRITE$MARK THEN CALL WRITE$RECORD;
C1132 5      CALL CLOSE(C$ADDR);
C1133 5      CALL INC$PTR(2);
C1134 5      END;
C1135 4
C1136 4      /* SER */
C1137 4
C1138 4      DC;
C1139 4      IF OVERFLOW THEN PROGRAM$CCOUNTER = C$ADDR;
C1140 5      ELSE CALL INC$PTR(2);
C1141 4      END;
C1142 4
C1143 4      /* BFN */
C1144 4      PROGRAM$CCOUNTER=C$ADDR;
C1145 4
C1146 4      /* CFN */
C1147 4      CALL GPEN$FILE(1);
C1148 4
C1149 4      /* CP1 */
C1150 4      CALL GPEN$FILE(2);
C1151 4
C1152 4      /* CP2 */
C1153 4      CALL DPEN$FILE(3);
C1154 4
C1155 4      /* RGT */
C1156 4      DC;
C1157 4      IF NOT SIGN2 THEN
C1158 5      BRANCH$FLAG=NOT BRANCH$FLAG;
C1159 5      CALL CONDITIONAL$BRANCH(0);
C1160 4      END;
C1161 4
C1162 4      /* RLT */
C1163 4      DC;
C1164 4      IF SIGN2 THEN
C1165 5      BRANCH$FLAG=NCT BRANCH$FLAG;
C1166 5      CALL CONDITIONAL$BRANCH(0);
C1167 4      END;
C1168 4
C1169 4      /* REG */
C1170 4      DC;
C1171 4      IF NOT ZERO$RESULT THEN
C1172 5      BRANCH$FLAG=NCT BRANCH$FLAG;
C1173 5      CALL CONDITIONAL$BRANCH(0);
C1174 4      END;
C1175 4
C1176 4      /* INV */
C1177 4      CALL INCREMENT$OR$BRANCH(INVALID);
C1178 4
C1179 4      /* EOR */
C1180 4      CALL INCREMENT$OR$BRANCH(ENC$OF$RECORD);
C1181 4
C1182 4      /* ACC */
C1183 4      CALL ACCEPT;
C1184 4
C1185 4      /* DIS */
C1186 4      CALL DISPLAY;
C1187 4
C1188 4      /* STC */
C1189 4      DC;
C1190 4      CALL DISPLAY;
C1191 5      CALL STOP;
C1192 5      END;
C1193 4
C1194 4      /* LCI */
C1195 4      DC;
C1196 4      C$ADDR(3)=CCONVERT$TOSHEX(C$ADDR,C$BYTE(2));
C1197 5      CALL INC$PTR(3);
C1198 4      END;
C1199 4
C1200 4      /* DEC */
C1201 4      DC;
C1202 4      IF C$ADDR<>0 THEN C$ADDR=C$ADDR-1;
C1203 5      IF C$ADDR=0 THEN PROGRAM$CCOUNTER=C$ADDR(1);
C1204 5      ELSE CALL INC$PTR(4);
C1205 4      END;
C1206 4
C1207 4      /* STC */
C1208 4      DC;
C1209 4      CALL STORE$NUMERIC;
C1210 5      CALL INC$PTR(4);
C1211 4      END;
C1212 4
C1213 4      /* S11 */
C1214 4      DC;
C1215 4      CALL STORE$NUMERIC;
C1216 5      CALL SET$ZONE(C$ADDR+C$BYTE(2)-1);
C1217 5      END;
C1218 4
C1219 4
C1220 4
C1221 4
C1222 4
C1223 4
C1224 4
C1225 4
C1226 4
C1227 4
C1228 4
C1229 4
C1230 4
C1231 5
C1232 5

```



```

01233 4
01234 4 /* S12 */
01235 4
01236 4
01237 4 DC;
01238 4 CALL STORE$NUMERIC;
01239 5 CALL SET$ZONE(C$ADDR);
01240 4
01241 4 /* S13 */
01242 4
01243 4 DC;
01244 4 CALL CHECK$DECIMAL;
01245 4 BASE=C$ADDR + C$BYTE(2) - 1;
01246 5 CALL STORE$AS$CHAR(C$BYTE(2) - 1);
01247 5 CALL SET$SIGN$SEP(C$ADDR + C$BYTE(2) - 1);
01248 4
01249 4 /* S14 */
01250 4
01251 4 DC;
01252 4 CALL CHECK$DECIMAL;
01253 4 BASE=C$ADDR + C$BYTE(2);
01254 5 CALL STORE$AS$CHAR(C$BYTE(2)-1);
01255 5 CALL SET$SIGN$SEP(C$ADDR);
01256 4
01257 4 /* S15 */
01258 4
01259 4 DC;
01260 4 CALL CHECK$DECIMAL;
01261 4 R2(9)=R2(9) OR SIGN2;
01262 5 CALL MOVE(R2 + 9 - C$BYTE(2), C$ADDR, C$BYTE(2));
01263 5 CALL INC$PTR(4);
01264 4
01265 4 /* LCD */
01266 4
01267 4 CALL LOAD$NUMSLIT;
01268 4
01269 4 /* LC1 */
01270 4
01271 4 CALL LOAD$NUMERIC;
01272 4
01273 4 /* LC2 */
01274 4
01275 4 DC;
01276 4 DECLARE I BYTE;
01277 4 HOLD=C$ADDR;
01278 5 IF CHECK$FOR$SIGN(CTR:=H$BYTE(1:=C$BYTE(2)-1)) THEN
01279 5 DC;
01280 5 CALL SET$LOAD(POSITIVE);
01281 5 I=I+1;
01282 5
01283 5 END;
01284 5 ELSE DO;
01285 5 CALL SET$LOAD(NEGATIVE);
01286 5 CALL LOAD$AS$CHAR(CTR-ZONE);
01287 5
01288 5 END;
01289 5 CALL LOAD$NUMBERS(C$ADDR, I);
01290 4
01291 4 /* LC3 */
01292 4
01293 4 DC;
01294 4 HOLD=C$ADDR;
01295 5 IF CHECK$FOR$SIGN(H$BYTE) THEN
01296 5 DC;
01297 5 CALL SET$LOAD(POSITIVE);
01298 5 CALL LOAD$NUMBERS(C$ADDR, C$BYTE(2));
01299 5
01300 5 END;
01301 5 ELSE DO;
01302 5 CALL SET$LOAD(NEGATIVE);
01303 5 CALL LOAD$NUMBERS(C$ADDR+1, C$BYTE(2)-1);
01304 5 CALL LOAD$AS$CHAR(H$BYTE-ZONE);
01305 5
01306 5 END;
01307 4
01308 4 /* LC4 */
01309 4
01310 4 DC;
01311 4 HOLD=C$ADDR;
01312 5 IF H$BYTE(C$BYTE(2) - 1) = '+' THEN
01313 5 CALL SET$LOAD(1);
01314 5 ELSE CALL SET$LOAD(0);
01315 5 CALL LOAD$NUMBERS(C$ADDR, C$BYTE(2) - 1);
01316 4
01317 4 /* LC5 */
01318 4
01319 4 DC;
01320 4 HOLD=C$ADDR;
01321 5 IF (H$BYTE='+') THEN CALL SET$LOAD(1);
01322 5 ELSE CALL SET$LOAD(0);
01323 5 CALL LOAD$NUMBERS(C$ADDR, C$BYTE(2)-1);
01324 4
01325 4 /* LC6 */
01326 4
01327 4 DC;
01328 4 DECLARE I BYTE;
01329 4 HOLD=C$ADDR;
01330 5 CALL SET$LOAD(H$BYTE(1:=C$BYTE(2)-1));
01331 5 BASE=BASE * 9 - I;
01332 5 DO CTR = 0 TO I;
01333 5 B$BYTE(CTR)=H$BYTE(CTR);
01334 5
01335 5 END;
01336 5 B$BYTE(CTR)=B$BYTE(CTR) AND OFOH;
01337 5 CALL INC$PTR(5);
01338 4
01339 4
01340 4
01341 4
01342 4
01343 4
01344 4
01345 4
01346 4
01347 4
01348 4
01349 4
01350 4
01351 4
01352 4
01353 4
01354 4
01355 4
01356 4
01357 4
01358 4
01359 4
01360 4
01361 4
01362 4
01363 4
01364 4
01365 4
01366 4
01367 4
01368 4
01369 4
01370 4
01371 4
01372 4
01373 4
01374 4
01375 4
01376 4
01377 4
01378 4
01379 4
01380 4
01381 4
01382 4
01383 4
01384 4
01385 4
01386 4
01387 4
01388 4
01389 4
01390 4
01391 4
01392 4
01393 4
01394 4
01395 4
01396 4
01397 4
01398 4
01399 4
01400 4
01401 4
01402 4
01403 4
01404 4
01405 4
01406 4
01407 4
01408 4
01409 4
01410 4
01411 4
01412 4
01413 4
01414 4
01415 4
01416 4
01417 4
01418 4
01419 4
01420 4
01421 4
01422 4
01423 4
01424 4
01425 4
01426 4
01427 4
01428 4
01429 4
01430 4
01431 4
01432 4
01433 4
01434 4
01435 4
01436 4
01437 4
01438 4
01439 4
01440 4
01441 4
01442 4
01443 4
01444 4
01445 4
01446 4
01447 4
01448 4
01449 4
01450 4
01451 4
01452 4
01453 4
01454 4
01455 4
01456 4
01457 4
01458 4
01459 4
01460 4
01461 4
01462 4
01463 4
01464 4
01465 4
01466 4
01467 4
01468 4
01469 4
01470 4
01471 4
01472 4
01473 4
01474 4
01475 4
01476 4
01477 4
01478 4
01479 4
01480 4
01481 4
01482 4
01483 4
01484 4
01485 4
01486 4
01487 4
01488 4
01489 4
01490 4
01491 4
01492 4
01493 4
01494 4
01495 4
01496 4
01497 4
01498 4
01499 4
01500 4
01501 4
01502 4
01503 4
01504 4
01505 4
01506 4
01507 4
01508 4
01509 4
01510 4
01511 4
01512 4
01513 4
01514 4
01515 4
01516 4
01517 4
01518 4
01519 4
01520 4
01521 4
01522 4
01523 4
01524 4
01525 4
01526 4
01527 4
01528 4
01529 4
01530 4
01531 4
01532 4
01533 4
01534 4
01535 4
01536 4
01537 4
01538 4
01539 4
01540 4
01541 4
01542 4
01543 4
01544 4
01545 4
01546 4
01547 4
01548 4
01549 4
01550 4
01551 4
01552 4
01553 4
01554 4
01555 4
01556 4
01557 4
01558 4
01559 4
01560 4
01561 4
01562 4
01563 4
01564 4
01565 4
01566 4
01567 4
01568 4
01569 4
01570 4
01571 4
01572 4
01573 4
01574 4
01575 4
01576 4
01577 4
01578 4
01579 4
01580 4
01581 4
01582 4
01583 4
01584 4
01585 4
01586 4
01587 4
01588 4
01589 4
01590 4
01591 4
01592 4
01593 4
01594 4
01595 4
01596 4
01597 4
01598 4
01599 4
01600 4
01601 4
01602 4
01603 4
01604 4
01605 4
01606 4
01607 4
01608 4
01609 4
01610 4
01611 4
01612 4
01613 4
01614 4
01615 4
01616 4
01617 4
01618 4
01619 4
01620 4
01621 4
01622 4
01623 4
01624 4
01625 4
01626 4
01627 4
01628 4
01629 4
01630 4
01631 4
01632 4
01633 4
01634 4
01635 4
01636 4
01637 4
01638 4
01639 4
01640 4
01641 4
01642 4
01643 4
01644 4
01645 4
01646 4
01647 4
01648 4
01649 4
01650 4
01651 4
01652 4
01653 4
01654 4
01655 4
01656 4
01657 4
01658 4
01659 4
01660 4
01661 4
01662 4
01663 4
01664 4
01665 4
01666 4
01667 4
01668 4
01669 4
01670 4
01671 4
01672 4
01673 4
01674 4
01675 4
01676 4
01677 4
01678 4
01679 4
01680 4
01681 4
01682 4
01683 4
01684 4
01685 4
01686 4
01687 4
01688 4
01689 4
01690 4
01691 4
01692 4
01693 4
01694 4
01695 4
01696 4
01697 4
01698 4
01699 4
01700 4
01701 4
01702 4
01703 4
01704 4
01705 4
01706 4
01707 4
01708 4
01709 4
01710 4
01711 4
01712 4
01713 4
01714 4
01715 4
01716 4
01717 4
01718 4
01719 4
01720 4
01721 4
01722 4
01723 4
01724 4
01725 4
01726 4
01727 4
01728 4
01729 4
01730 4
01731 4
01732 4
01733 4
01734 4
01735 4
01736 4
01737 4
01738 4
01739 4
01740 4
01741 4
01742 4
01743 4
01744 4
01745 4
01746 4
01747 4
01748 4
01749 4
01750 4
01751 4
01752 4
01753 4
01754 4
01755 4
01756 4
01757 4
01758 4
01759 4
01760 4
01761 4
01762 4
01763 4
01764 4
01765 4
01766 4
01767 4
01768 4
01769 4
01770 4
01771 4
01772 4
01773 4
01774 4
01775 4
01776 4
01777 4
01778 4
01779 4
01780 4
01781 4
01782 4
01783 4
01784 4
01785 4
01786 4
01787 4
01788 4
01789 4
01790 4
01791 4
01792 4
01793 4
01794 4
01795 4
01796 4
01797 4
01798 4
01799 4
01800 4
01801 4
01802 4
01803 4
01804 4
01805 4
01806 4
01807 4
01808 4
01809 4
01810 4
01811 4
01812 4
01813 4
01814 4
01815 4
01816 4
01817 4
01818 4
01819 4
01820 4
01821 4
01822 4
01823 4
01824 4
01825 4
01826 4
01827 4
01828 4
01829 4
01830 4
01831 4
01832 4
01833 4
01834 4
01835 4
01836 4
01837 4
01838 4
01839 4
01840 4
01841 4
01842 4
01843 4
01844 4
01845 4
01846 4
01847 4
01848 4
01849 4
01850 4
01851 4
01852 4
01853 4
01854 4
01855 4
01856 4
01857 4
01858 4
01859 4
01860 4
01861 4
01862 4
01863 4
01864 4
01865 4
01866 4
01867 4
01868 4
01869 4
01870 4
01871 4
01872 4
01873 4
01874 4
01875 4
01876 4
01877 4
01878 4
01879 4
01880 4
01881 4
01882 4
01883 4
01884 4
01885 4
01886 4
01887 4
01888 4
01889 4
01890 4
01891 4
01892 4
01893 4
01894 4
01895 4
01896 4
01897 4
01898 4
01899 4
01900 4
01901 4
01902 4
01903 4
01904 4
01905 4
01906 4
01907 4
01908 4
01909 4
01910 4
01911 4
01912 4
01913 4
01914 4
01915 4
01916 4
01917 4
01918 4
01919 4
01920 4
01921 4
01922 4
01923 4
01924 4
01925 4
01926 4
01927 4
01928 4
01929 4
01930 4
01931 4
01932 4
01933 4
01934 4
01935 4
01936 4
01937 4
01938 4
01939 4
01940 4
01941 4
01942 4
01943 4
01944 4
01945 4
01946 4
01947 4
01948 4
01949 4
01950 4
01951 4
01952 4
01953 4
01954 4
01955 4
01956 4
01957 4
01958 4
01959 4
01960 4
01961 4
01962 4
01963 4
01964 4
01965 4
01966 4
01967 4
01968 4
01969 4
01970 4
01971 4
01972 4
01973 4
01974 4
01975 4
01976 4
01977 4
01978 4
01979 4
01980 4
01981 4
01982 4
01983 4
01984 4
01985 4
01986 4
01987 4
01988 4
01989 4
01990 4
01991 4
01992 4
01993 4
01994 4
01995 4
01996 4
01997 4
01998 4
01999 4
02000 4

```



```

01341 4
01342 4 /* FER */
01343 4
01344 4 DC;
01345 4     BASE=C$ADDR(1)+1;
01346 4     B$ADDR=C$ADDR(2);
01347 5     PROGRAM$COUNTER=C$ADDR;
01348 5
01349 4 END;
01350 4
01351 4 /* CAL */
01352 4
01353 4 CALL CCMP$NUM$UNSIGNED;
01354 4
01355 4 /* CNS */
01356 4
01357 4 CALL CCMP$NUM$SIGN;
01358 4
01359 4 /* CAL */
01360 4
01361 4 CALL CCMP$ALPHA;
01362 4
01363 4 /* RLS */
01364 4
01365 4 DC;
01366 5 CALL BACK$CNE$RECORD;
01367 5 CALL WRITESFROM$MEMORY;
01368 5 CALL INC$PTR(6);
01369 4
01370 4 END;
01371 4
01372 4 /* CLS */
01373 4
01374 4 DC;
01375 5 CALL BACK$CNE$RECORD;
01376 5 CALL WRITESZERO$RECORD;
01377 5 CALL INC$PTR(6);
01378 4
01379 4 END;
01380 4
01381 4 /* RCF */
01382 4
01383 4 DC;
01384 5 CALL READ$TO$MEMORY;
01385 5 CALL INC$PTR(6);
01386 4
01387 4 END;
01388 4
01389 4 /* WTF */
01390 4
01391 4 DC;
01392 5 CALL WRITESFROM$MEMORY;
01393 5 CALL INC$PTR(6);
01394 4
01395 4 END;
01396 4
01397 4 /* RVL */
01398 4
01399 4 CALL READ$VARIABLE;
01400 4
01401 4 /* WVL */
01402 4
01403 4 CALL WRITE$VARIABLE;
01404 4
01405 4 /* SCR */
01406 4
01407 4 DC;
01408 5 SUBSCRIPT(C$BYTE(2))=
01409 5     CONVERT$TO$HEX(C$ADDR,C$BYTE(3));
01410 5 CALL INC$PTR(4);
01411 4
01412 4 END;
01413 4
01414 4 /* SGT */
01415 4
01416 4 CALL STRING$CCMPARE(1);
01417 4
01418 4 /* SLT */
01419 4
01420 4 CALL STRING$CCMPARE(0);
01421 4
01422 4 /* SEC */
01423 4
01424 4 CALL STRING$COMPARE(2);
01425 4
01426 4 /* MCV */
01427 4
01428 4 DC;
01429 5 CALL MOVE(RES(C$ADDR(1)),RES(C$ADDR),C$ADDR(2));
01430 5 IF C$ADDR(3)>0 THEN CALL
01431 5     FILL(RES(C$ADDR(1))+C$ADDR(2),C$ADDR(3),' ');
01432 5 CALL INC$PTR(8);
01433 4
01434 4 END;
01435 4
01436 4 /* RRS */
01437 4
01438 4 DC;
01439 5 CALL READ$TO$MEMORY;
01440 5 CALL GET$REC$NUMBER;
01441 5 CALL INC$PTR(9);
01442 4
01443 4 END;
01444 4
01445 4 /* WRS */
01446 4
01447 4 DC;
01448 5 CALL WRITESFROM$MEMORY;
01449 5 CALL GET$REC$NUMBER;
01450 5 CALL INC$PTR(9);
01451 4
01452 4 END;

```



```

00001 1
00002 1      /* THIS PROGRAM TAKES THE CODE OUTPUT FROM THE COBOL COMPILER
00003 1      AND BUILDS THE ENVIRONMENT FOR THE COBOL INTERPRETER */
00004 1
00005 1      10CH:      /* LOAD PCINT */
00006 1
00007 1      DECLARE
00008 1
00009 1          LIT      LITERALLY      'LITERALLY',
00010 1          BCCT      LIT      '0',
00011 1          BCCS      LIT      '5',
00012 1          TRUE      LIT      '1',
00013 1          FALSE      LIT      '0',
00014 1          FOREVER      LIT      'WHILE TRUE',
00015 1          FCB      ADDRESS      INITIAL (5CH),
00016 1          FCB$BYTE      BASED      FCB      BYTE,
00017 1          I      BYTE,
00018 1          ADDR      ADDRESS      INITIAL (100H),
00019 1          CHAR      BASED      ADDR      BYTE,
00020 1          BUFF$END      LIT      '100H',
00021 1          INTERP$FCB      (23)      BYTE      INITIAL(0,'CINTERP COM',0,0,C,C),
00022 1          CODE$NOT$SET      BYTE      INITIAL (TRUE),
00023 1          READER$LOCATION      LIT      '1C80H',
00024 1          INTERP$ADDRESS      ADDRESS      INITIAL(2000H),
00025 1          INTERP$CONTENT      BASED      INTERP$ADDRESS, ADDRESS,
00026 1          I$BYTE      BASED      INTERP$ADDRESS      BYTE,
00027 1          CODE$CTR      ADDRESS,
00028 1          C$BYTE      BASED      CODE$CTR      BYTE,
00029 1          BASE      ADDRESS,
00030 1          B$ADDR      BASED      BASE      ADDRESS,
00031 1          B$BYTE      BASED      BASE      BYTE;
00032 1
00033 1      MCN1: PROCEDURE (F,A);
00034 2          DECLARE F BYTE, A ADDRESS;
00035 2          GC TO BCCS;
00036 2      END MCN1;
00037 1
00038 1      MCN2: PROCEDURE (F,A) BYTE;
00039 2          DECLARE F BYTE, A ADDRESS;
00040 2          GC TO BCCS;
00041 2      END MCN2;
00042 1
00043 1      PRINT$CHAR: PROCEDURE(CHAR);
00044 2          DECLARE CHAR BYTE;
00045 2          CALL MCN1(2,CHAR);
00046 2      END PRINT$CHAR;
00047 1
00048 1      CRLF: PROCEDURE;
00049 2          CALL PRINT$CHAR(13);
00050 2          CALL PRINT$CHAR(10);
00051 2      END CRLF;
00052 1
00053 1      PRINT: PROCEDURE(A);
00054 2          DECLARE A ADDRESS;
00055 2          CALL CRLF;
00056 2          CALL MCN1(9,A);
00057 2      END PRINT;
00058 1
00059 1      CFEN: PROCEDURE (A) BYTE;
00060 2          DECLARE A ADDRESS;
00061 2          RETURN MCN2(15,A);
00062 2      END CFEN;
00063 1
00064 1      MOVE: PROCEDURE(FROM, DEST, COUNT);
00065 2          DECLARE (FROM, DEST, COUNT) ADDRESS,
00066 2          (F BASED FROM, D BASED DEST) BYTE;
00067 2          DO WHILE (COUNT=CCOUNT-1)<>0FFFFH;
00068 3              D=F;
00069 3              FROM=FROM+1;
00070 3              DEST=DEST+1;
00071 3          END;
00072 2      END MOVE;
00073 1
00074 1      GET$CHAR: PROCEDURE BYTE;
00075 2          IF (ADDR:=ADDR + 1)>=BUFF$END THEN
00076 2              CC;
00077 2              IF MCN2(20,FCB)<>0 THEN
00078 3                  CC;
00079 3                  CALL PRINT('END OF INPUT  $');
00080 3                  GC TO BCCT;
00081 3              END;
00082 2              ADDR=BCF;
00083 2          END;
00084 2          RETURN CHAR;
00085 2      END GET$CHAR;
00086 1
00087 1
00088 1
00089 1
00090 1
00091 1
00092 1
00093 1

```



```

CC094 1
CC095 1 NEXT$CHAR: PROCEDURE;
CC096 2 CHAR=GET$CHAR;
CC097 2 END NEXT$CHAR;
CC098 1
CC099 1
CC100 1 STORE: PROCEDURE(CCUNT);
CC101 2 DECLARE (CCUNT BYTE;
CC102 2 IF CODE$ACT$SET THEN
CC103 2 DO;
CC104 3 CALL PRINT('CODE ERRORS');
CC105 3 CALL NEXT$CHAR;
CC106 3 RETURN;
CC107 2 END;
CC108 2 CC I=1 TO CCUNT;
CC109 3 C$BYTE=CHAR;
CC110 3 CALL NEXT$CHAR;
CC111 3 CODE$CTR=CODE$CTR+1;
CC112 2 END;
CC113 2 END STORE;
CC114 1
CC115 1
CC116 1 BACK$STUFF: PROCEDURE;
CC117 2 DECLARE (HCLC,STUFF) ADDRESS;
CC118 2 BASE=HCLC;
CC119 2 CC I=0 TO 3;
CC120 3 B$BYTE(I)=GET$CHAR;
CC121 2 END;
CC122 2 CC FOREVER;
CC123 3 BASE=HCLC;
CC124 3 HCLC=B$ADDR;
CC125 3 B$ADDR=STUFF;
CC126 3 IF HCLC=0 THEN
CC127 3 DO;
CC128 4 CALL NEXT$CHAR;
CC129 4 RETURN;
CC130 3 END;
CC131 2 END;
CC132 2 END BACK$STUFF;
CC133 1
CC134 1
CC135 1 START$CODE: PROCEDURE;
CC136 2 CODE$NOT$SET=FALSE;
CC137 2 I$BYTE=GET$CHAR;
CC138 2 I$BYTE(I)=GET$CHAR;
CC139 2 CODE$CTR=INTERP$CCNTENT;
CC140 2 CALL NEXT$CHAR;
CC141 2 END START$CODE;
CC142 1
CC143 1
CC144 1 GC$DEPENDING: PROCEDURE;
CC145 2 CALL STORE(1);
CC146 2 CALL STORE(2*H(CHAR,1) + 4);
CC147 2 END GC$DEPENDING;
CC148 1
CC149 1
CC150 1 INITIALIZE: PROCEDURE;
CC151 2 DECLARE (CCUNT,WHERE,HOW$MANY) ADDRESS;
CC152 2 BASE=WHERE;
CC153 2 CC I=0 TO 3;
CC154 3 B$BYTE(I)=GET$CHAR;
CC155 2 END;
CC156 2 BASE=WHERE - 1;
CC157 2 CC COUNT = 1 TO HOW$MANY;
CC158 3 B$BYTE(CCUNT)=GET$CHAR;
CC159 2 END;
CC160 2 CALL NEXT$CHAR;
CC161 2 END INITIALIZE;
CC162 1

```



```

00163 1
00164 1 BUILD: PROCEDURE;
00165 2 DECLARE
00166 2 F2 LIT 'B';
00167 2 F3 LIT 'S';
00168 2 F4 LIT 'N';
00169 2 F5 LIT 'I';
00170 2 F6 LIT 'A';
00171 2 F7 LIT 'S';
00172 2 F8 LIT 'S';
00173 2 F9 LIT 'S';
00174 2 F10 LIT 'A';
00175 2 F11 LIT 'C';
00176 2 F13 LIT 'I';
00177 2 CCP LIT '2';
00178 2 INT LIT '3';
00179 2 EST LIT '4';
00180 2 TER LIT '5';
00181 2 SCC LIT '6';
00182 2
00183 2 CC FOREVER;
00184 2 IF CHAR < F2 THEN CALL STORE(1);
00185 2 ELSE IF CHAR < F3 THEN CALL STORE(2);
00186 2 ELSE IF CHAR < F4 THEN CALL STORE(3);
00187 2 ELSE IF CHAR < F5 THEN CALL STORE(4);
00188 2 ELSE IF CHAR < F6 THEN CALL STORE(5);
00189 2 ELSE IF CHAR < F7 THEN CALL STORE(6);
00190 2 ELSE IF CHAR < F8 THEN CALL STORE(7);
00191 2 ELSE IF CHAR < F9 THEN CALL STORE(8);
00192 2 ELSE IF CHAR < F10 THEN CALL STORE(9);
00193 2 ELSE IF CHAR < F11 THEN CALL STORE(10);
00194 2 ELSE IF CHAR < F13 THEN CALL STORE(11);
00195 2 ELSE IF CHAR = GDP THEN CALL STORE(13);
00196 2 ELSE IF CHAR = GDP THEN CALL GC$C$PENDING;
00197 2 ELSE IF CHAR = BST THEN CALL BACK$STUFF;
00198 2 ELSE IF CHAR = INT THEN CALL INITIALIZE;
00199 2 ELSE IF CHAR = TER THEN
00200 2 CALL PRINT('LOAD FINISHED$');
00201 2 RETURN;
00202 2
00203 2 END;
00204 2 ELSE IF CHAR = SCC THEN CALL START$CODE;
00205 2 CC;
00206 2 IF CHAR <> OFFH THEN CALL PRINT('LOAD ERRCR$');
00207 2 CALL NEXT$CHAR;
00208 2
00209 2 END;
00210 2
00211 2 END BUILD;
00212 2
00213 2 /* PROGRAM EXECUTION STARTS HERE */
00214 2
00215 2 FCB$BYTE=0;
00216 2 CALL MCVE('CIN',0,C,0,0),FCB + 9,7);
00217 2 IF OPEN(FCB)=255 THEN
00218 2 CO;
00219 2 CALL PRINT('FILE NCT FOUND $');
00220 2 GC TO BCCT;
00221 2
00222 2 END;
00223 2 CALL NEXT$CHAR;
00224 2 CALL BUILD;
00225 2 CALL MCVE('INTERP$FCB,FCB,33);
00226 2 IF OPEN(FCB)=255 THEN
00227 2 CO;
00228 2 CALL PRINT('INTERPRETER NCT FOUND $');
00229 2 GC TO BCCT;
00230 2
00231 2 END;
00232 2 CALL MCVE(READER$LOCATICN, 80H, 80H);
00233 2 GO TO 80H;
00234 2
00235 2 EQF

```



```

00001 1
00002 1
00003 1
00004 1
00005 1
00006 1
00007 1
00008 1
00009 1
00010 1
00011 1
00012 1
00013 1
00014 1
00015 1
00016 1
00017 1
00018 1
00019 1
00020 1
00021 1
00022 2
00023 2
00024 2
00025 1
00026 1
00027 1
00028 2
00029 2
00030 2
00031 1
00032 1
00033 1
00034 2
00035 2
00036 2
00037 1
00038 1
00039 1
00040 2
00041 2
00042 2
00043 1
00044 1
00045 1
00046 2
00047 2
00048 2
00049 2
00050 3
00051 3
00052 2
00053 1
00054 1
00055 1
00056 2
00057 2
00058 3
00059 3
00060 4
00061 4
00062 4
00063 3
00064 3
00065 2
00066 2
00067 1
00068 1
00069 1
00070 1
00071 2
00072 2
00073 1
00074 1
00075 1
00076 1
00077 2
00078 2
00079 2
00080 2
00081 2
00082 2
00083 2
00084 2
00085 2
00086 2
00087 2
00088 2
00089 2
00090 2
00091 2
00092 2
00093 2
00094 2
00095 2
00096 2
00097 2
00098 2
00099 2
00100 2
00101 2
00102 2
00103 2

```

```

/* THIS PROGRAM TAKES THE COOE OUTPUT FROM THE COBCL COMPILER
AND CONVERTS IT INTO A READABLE OUTPUT TO FACILITATE DEBUGGING */

100H:      /* LOAD POINT */

DECLARE

LIT          LITERALLY      'LITERALLY',
BCCT         LIT            '0',
BOCS         LIT            '5',
FCB          ADDRESS        INITIAL (5CH),
FCB$BYTE     BASED          FCB BYTE,
I            BYTE,
ACOR         ADDRESS        INITIAL (100H),
CHAR         BASED          ADDR BYTE,
C$ACOR       BASED ACOR      ADDRESS,
BUFF$ENO     LIT            'OFFH',
FILE$TYPE    DATA ('C','I','N');

MCN1: PROCEDURE (F,A);
DECLARE F BYTE, A ADDRESS;
GC TO BCCT;
END MCN1;

MCN2: PROCEDURE (F,A) BYTE;
DECLARE F BYTE, A ADDRESS;
GC TO BCCT;
END MCN2;

PRINT$CHAR: PROCEDURE (CHAR);
DECLARE CHAR BYTE;
CALL MCN1(2,CHAR);
END PRINT$CHAR;

CRLF: PROCEDURE;
CALL PRINT$CHAR(13);
CALL PRINT$CHAR(10);
END CRLF;

P: PROCEDURE (ADD1);
DECLARE ADD1 ADDRESS, C BASED A001 BYTE;
CALL CRLF;
CC 1=0 TO 2;
CALL PRINT$CHAR(C(1));
END;
CALL PRINT$CHAR(' ');
END P;

GET$CHAR: PROCEDURE BYTE;
IF (ACOR=ACOR + 1)>BUFF$ENO THEN
CC;
IF MCN2(20,FCB)<>0 THEN
CC;
CALL P('ENO');
CALL TIME(10);
GC TO BCCT;
END;
ACOR=BCF;
END;
RETURN CHAR;
END GET$CHAR;

O$CHAR: PROCEDURE (OUTPUT$BYTE);
DECLARE OUTPUT$BYTE BYTE;
IF OUTPUT$BYTE<10 THEN CALL PRINT$CHAR(OUTPUT$BYTE + 20H);
ELSE CALL PRINT$CHAR(OUTPUT$BYTE + 37H);
END O$CHAR;

D: PROCEDURE (CCLNT);
DECLARE (CCLNT,J) ADDRESS;
DC J=1 TO CCLNT;
CALL D$CHAR(SHR(GET$CHAR,4));
CALL D$CHAR(CHAR AND 0FH);
CALL PRINT$CHAR(' ');
END;
END D;

PRINT$REST: PROCEDURE;
DECLARE
F5         LIT            'E',
F4         LIT            'C',
F3         LIT            'I',
F2         LIT            'T',
F1         LIT            'E',
F6         LIT            'S',
F7         LIT            'I',
F8         LIT            'Z',
F9         LIT            'E',
F10        LIT            'S',
F11        LIT            'E',
F12        LIT            'S',
F13        LIT            'E',
GDP        LIT            'E',
INT        LIT            'E',
AST        LIT            'E',
TER        LIT            'E',
SCO        LIT            'E';

```



```

00104 2 IF CHAR < F2 THEN RETURN;
00105 NNN IF CHAR < F3 THEN DO: CALL D(1); RETURN; END;
00106 NNN IF CHAR < F4 THEN DO: CALL D(2); RETURN; END;
00107 NNN IF CHAR < F5 THEN DO: CALL D(3); RETURN; END;
00108 NNN IF CHAR < F6 THEN DO: CALL D(4); RETURN; END;
00109 NNN IF CHAR < F7 THEN DO: CALL D(5); RETURN; END;
00110 NNN IF CHAR < F8 THEN DO: CALL D(6); RETURN; END;
00111 NNN IF CHAR < F9 THEN DO: CALL D(7); RETURN; END;
00112 NNN IF CHAR < F10 THEN DO: CALL D(8); RETURN; END;
00113 NNN IF CHAR < F11 THEN DO: CALL D(9); RETURN; END;
00114 NNN IF CHAR < F12 THEN DO: CALL D(10); RETURN; END;
00115 NNN IF CHAR < F13 THEN DO: CALL D(11); RETURN; END;
00116 NNN IF CHAR < F14 THEN DO: CALL D(12); RETURN; END;
00117 NNN IF CHAR = GCF THEN DO: CALL D(1); CALL D(SHL(CHAR,1)+5); RETURN; END;
00118 NNN IF CHAR = INT THEN DO: CALL D(3); CALL D(C$ADDR + 1); RETURN; END;
00119 NNN IF CHAR = BST THEN DO: CALL D(4); RETURN; END;
00120 NNN IF CHAR = TER THEN DO: CALL P('END'); GO TO BOOT; END;
00121 NNN IF CHAR = SCC THEN DO: CALL D(2); RETURN; END;
00122 NNN IF CHAR <> CFFH THEN CALL P('XXX');
00123 1 END PRINT$REST;
00124 1
00125 1 /* PROGRAM EXECUTION STARTS HERE */
00126 1 FCB$BYTE=0;
00127 1 DC I=C TO 2;
00128 1 FCB$BYTE(I+9)=FILE$TYPE(I);
00129 1 END;
00130 1 IF MCN2(15,FCB)=255 THEN DO: CALL P('ZZZ'); GC TO BOOT; END;
00131 1
00132 1 DO WHILE 1;
00133 1 IF GET$CHAR <= 66 THEN DO CASE CHAR;
00134 1 NNN /* CASE 0 NOT USED */
00135 1 NNN CALL P('ADD');
00136 1 NNN CALL P('SUB');
00137 1 NNN CALL P('MUL');
00138 1 NNN CALL P('DIV');
00139 1 NNN CALL P('NEG');
00140 1 NNN CALL P('ST1');
00141 1 NNN CALL P('ST2');
00142 1 NNN CALL P('ST3');
00143 1 NNN CALL P('ST4');
00144 1 NNN CALL P('ST5');
00145 1 NNN CALL P('ST6');
00146 1 NNN CALL P('ST7');
00147 1 NNN CALL P('ST8');
00148 1 NNN CALL P('ST9');
00149 1 NNN CALL P('ST10');
00150 1 NNN CALL P('ST11');
00151 1 NNN CALL P('ST12');
00152 1 NNN CALL P('ST13');
00153 1 NNN CALL P('ST14');
00154 1 NNN CALL P('ST15');
00155 1 NNN CALL P('ST16');
00156 1 NNN CALL P('ST17');
00157 1 NNN CALL P('ST18');
00158 1 NNN CALL P('ST19');
00159 1 NNN CALL P('ST20');
00160 1 NNN CALL P('ST21');
00161 1 NNN CALL P('ST22');
00162 1 NNN CALL P('ST23');
00163 1 NNN CALL P('ST24');
00164 1 NNN CALL P('ST25');
00165 1 NNN CALL P('ST26');
00166 1 NNN CALL P('ST27');
00167 1 NNN CALL P('ST28');
00168 1 NNN CALL P('ST29');
00169 1 NNN CALL P('ST30');
00170 1 NNN CALL P('ST31');
00171 1 NNN CALL P('ST32');
00172 1 NNN CALL P('ST33');
00173 1 NNN CALL P('ST34');
00174 1 NNN CALL P('ST35');
00175 1 NNN CALL P('ST36');
00176 1 NNN CALL P('ST37');
00177 1 NNN CALL P('ST38');
00178 1 NNN CALL P('ST39');
00179 1 NNN CALL P('ST40');
00180 1 NNN CALL P('ST41');
00181 1 NNN CALL P('ST42');
00182 1 NNN CALL P('ST43');
00183 1 NNN CALL P('ST44');
00184 1 NNN CALL P('ST45');
00185 1 NNN CALL P('ST46');
00186 1 NNN CALL P('ST47');
00187 1 NNN CALL P('ST48');
00188 1 NNN CALL P('ST49');
00189 1 NNN CALL P('ST50');
00190 1 NNN CALL P('ST51');
00191 1 NNN CALL P('ST52');
00192 1 NNN CALL P('ST53');
00193 1 NNN CALL P('ST54');
00194 1 NNN CALL P('ST55');
00195 1 NNN CALL P('ST56');
00196 1 NNN CALL P('ST57');
00197 1 NNN CALL P('ST58');
00198 1 NNN CALL P('ST59');
00199 1 NNN CALL P('ST60');
00200 1 NNN CALL P('ST61');
00201 1 NNN CALL P('ST62');
00202 1 NNN END; /* CF CASE STATEMENT */
00203 1 CALL PRINT$REST;
00204 1 END; /* END CF CC WHILE */
00205 1 EOF

```



```

01444 4
01445 4 /* RRR */
01446 4
01447 4 CC;
01448 4 CALL SET$RANDOM$POINTER;
01449 4 CALL READ$TO$MEMORY;
01450 5 CALL INC$PTR(9);
01451 5
01452 4 /* WRR */
01453 4
01454 4 CALL WRITE$RANDOM;
01455 4
01456 4 /* RRR */
01457 4
01458 4 CALL WRITE$RANDOM;
01459 4
01460 4
01461 4 /* CLR */
01462 4
01463 4 CC;
01464 4 CALL SET$RANDOM$POINTER;
01465 5 CALL WRITE$ZERO$RECORD;
01466 5 CALL INC$PTR(9);
01467 5
01468 4 /* MED */
01469 4
01470 4
01471 4 CC;
01472 4 CALL MOVE(C$ADDR(3),C$ADDR,C$ADDR(4));
01473 5 BASE=C$ADDR(1);
01474 5 HOLD=C$ALDR;
01475 5 CTR=0;
01476 5 DO WHILE (CTR<C$ADDR(1))AND(CTR<C$ADDR(4));
01477 5 CALL CHECK$EDIT(H$BYTE);
01478 6 END;
01479 5 IF CTR < C$ADDR(4) THEN
01480 5 CALL FILL(HOLD,C$ADDR(4)-CTR,' ');
01481 5
01482 4 /* MNE */
01483 4
01484 4 ;
01485 4
01486 4 /* GCP */
01487 4
01488 4
01489 4 CC;
01490 4 DECLARE OFFSET BYTE;
01491 5 OFFSET=CONVERT$TO$HEX(C$ADDR(1),C$BYTE(1)-1);
01492 5 IF OFFSET > C$BYTE + 1 THEN
01493 5 DO;
01494 5 CALL PRINT$ERROR('GC');
01495 6 CALL INC$PTR(SHL(C$BYTE,1) + 6);
01496 6
01497 5 END;
01498 5 ELSE PROGRAM$COUNTER=C$ADDR(OFFSET + 2);
01499 4
01500 4
01501 3 END; /* END OF CASE STATEMENT */
01502 2 END EXECUTE;
01503 1
01504 1 /* * * * * * PROGRAM EXECUTION STARTS HERE * * * * * */
01505 1
01506 1 BASE=CODE$START;
01507 1 PROGRAM$COUNTER=B$ADDR;
01508 1 CALL EXECUTE;
01509 1 EOF

```



```

00001 1      /* COBCL CCMFILE - PART 2 READER */
00002 1
00003 1
00004 1      /* THIS PROGRAM IS LOADED IN WITH THE PART 1 PROGRAM
00005 1      AND IS CALLED WHEN PART 1 IS FINISHED. THIS PROGRAM
00006 1      OPENS THE PART2.CCM FILE THAT CONTAINS THE CODE FOR
00007 1      PART 2 OF THE COMPILER, AND READS IT INTO CCCE. AT
00008 1      THE END OF THE READ OPERATION, CCNTRL IS PASSED TO
00009 1      THE SECOND PART PROGRAM. */
00010 1
00011 1
00012 1      31COH:  /* LCAC POINT */
00013 1
00014 1      DECLARE
00015 1
00016 1      BCCT      LITERALLY '0H', /* ENTRY TO THE OPERATING SYSTEM */
00017 1      BCOS      LITERALLY '5F', /* STARTING LOCATION FOR PASS 2 */
00018 1      START     LITERALLY '1COH', /* STARTING LOCATION FOR PASS 2 */
00019 1      FCB(33)   BYTE INITIAL(0,'PASS2.CCM',0,0,0,0),
00020 1      LASTDMA   ADDRESS  INITIAL(2480H), /* 80 LESS THAN MEMCRY */
00021 1      I         ADDRESS;
00022 1
00023 1      MCNA: PROCEDURE(F,A);
00024 1      DECLARE F BYTE, A ADDRESS;
00025 1      GC TO BCCT;
00026 1      END MCNA;
00027 1
00028 1      MCNB: PROCEDURE(F,A)BYTE;
00029 1      DECLARE F BYTE, A ADDRESS;
00030 1      GC TO BCOS;
00031 1      ENO MCNB;
00032 1
00033 1      ERROR: PROCEDURE(CODE);
00034 1      DECLARE CCCE ADDRESS;
00035 1      CALL MCNA(2,(HIGH(CCODE)));
00036 1      CALL MCNA(2,(LOW(CCODE)));
00037 1      CALL TIME(1);
00038 1      GC TO BCCT;
00039 1      END ERROR;
00040 1
00041 1      /* OPEN PASS2.CCM */
00042 1      IF MONE(15,.FCB)=255 THEN CALL ERRCR('02');
00043 1      /* READ IN FILE */
00044 1      CO I=100H TO LASTDMA BY 80H;
00045 1      CALL MCNA(26,I); /* SET DMA */
00046 1      IF MONB(20,.FCB)<>0 THEN CALL ERRCR('R2');
00047 1      ENO;
00048 1      CALL MCNA(26,8CH); /* RESET DMA */
00049 1      GC TO START;
00050 1      EOF

```

```

00001 1      /* COBCL CCMFILE - INTERP READER */
00002 1
00003 1
00004 1      /* THIS PROGRAM IS CALLED BY THE BUILD PROGRAM AFTER
00005 1      CBLINT.CCM HAS BEEN OPENED, AND READS THE CCCE INTO MEMCRY
00006 1      */
00007 1
00008 1
00009 1      80H:  /* LCAC POINT */
00010 1
00011 1      DECLARE
00012 1
00013 1      BCCT      LITERALLY '0H', /* ENTRY TO THE OPERATING SYSTEM */
00014 1      BCOS      LITERALLY '5F', /* STARTING LOCATION FOR PASS 2 */
00015 1      START     LITERALLY '1COH', /* STARTING LOCATION FOR PASS 2 */
00016 1      LASTDMA   ADDRESS  INITIAL(1E80H), /* 80 LESS THAN MEMCRY */
00017 1      I         ADDRESS;
00018 1
00019 1      MCNA: PROCEDURE(F,A);
00020 1      DECLARE F BYTE, A ADDRESS;
00021 1      GC TO BCCT;
00022 1      ENO MCNA;
00023 1
00024 1      MCNB: PROCEDURE(F,A)BYTE;
00025 1      DECLARE F BYTE, A ADDRESS;
00026 1      GC TO BCOS;
00027 1      END MCNB;
00028 1
00029 1      GO I=1COH TO LASTDMA BY 80H;
00030 1      CALL MCNA(26,I); /* SET DMA */
00031 1      IF MONB(20,5CH)<>0 THEN GO TO BOOT;
00032 1      ENO;
00033 1      GC TO START;
00034 1      EOF

```


LIST OF REFERENCES

1. American National Standards Institute, COBOL Standard, ANSI X3.23-1974.
2. Aho, A. V. and S. C. Johnson, LR Parsing, Computing Surveys, Vol. 6 No. 2, June 1974.
3. Bauer, F. L. and J. Eickel, editors, Compiler Construction - An Advanced Course, Lecture notes in Computer Science, Springer-Verlag, New York 1976.
4. Digital Research, An Introduction to CP/M Features and Facilities, 1976.
5. Digital Research, CP/M Interface Guide, 1976.
6. Eubanks, Gordon E. Jr. A Microprocessor Implementation of Extended Basic, Masters Thesis, Naval Postgraduate School, December 1976.
7. Intel Corporation, 8008 and 8080 PL/M Programming Manual, 1975.
8. Intel Corporation, 8080 Simulator Software Package, 1974.
9. Knuth, Donald E. On the Translation of Languages from Left to Right, Information and Control Vol. 6, No. 6, 1965.
10. Software Development Division, ADPE Selection Office, Department of the Navy, NYPD-CUBOL, April 1975.
11. Strutynski, Kathryn B. Information on the CP/M Interface Simulator, internally distributed technical note.
12. University of Toronto, Computer Systems Research Group

Technical Report CSPG-2, "An Efficient LALR Parser
Generator," by W. R. Lalonge, April 1971.

INITIAL DISTRIBUTION LIST

	No. Copies
1. Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2. Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3. Department Chairman, Code 52 Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
4. Assoc Professor G. A. Kildall, Code 52Kd Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
5. Lt L. V. Rich, Code 52Rs Department of Computer Science Naval Postgraduate School Monterey, California 93940	1
6. ADPE Selection Office Department of the Navy Washington, D. C. 20376	1
7. Capt A. S. Craig, USMC 611 Canyon Drive, Springville, Utah 84663	1

Thesis
C78215 Craig
c.1

169864

MICRO-COBOL an im-
plementation of Navy
standard HYPO-COBOL
for a microprocessor-
based computer system.

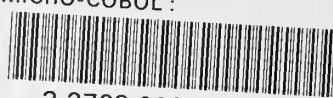
Thesis
C78215 Craig
c.1

169864

MICRO-COBOL an im-
plementation of Navy
standard HYPO-COBOL
for a microprocessor-
based computer system.



COBOL :



3 2768 002 09925 1
DUDLEY KNOX LIBRARY